O'REILLY

TURING 图灵程序设计丛书

探索JavaScript语言核心概念

深入了解ES6，展望JavaScript发展方向

[美] KYLE SIMPSON 著

单业 译

你不知道的JavaScript 下卷

UP & GOING

ES6 & BEYOND

YOU DON'T KNOW JS

中国工信出版集团 人民邮电出版社 POSTS & TELECOM PRESS

数字版权声明

图灵社区的电子书没有采用专有客户端，您可以在任意设备上，用自己喜欢的浏览器和PDF阅读器进行阅读。

但您购买的电子书仅供您个人使用，未经授权，不得进行传播。

我们愿意相信读者具有这样的良知和觉悟，与我们共同保护知识产权。

如果购买者有侵权行为，我们可能对该用户实施包括但不限于关闭该帐号等维权措施，并可能追究法律责任。

译者介绍

单业

计算机专业硕士，软件工程师，曾供职于多家软件公司，从事软件开发工作，现居于上海。译有《你不知道的JavaScript（中卷）》。

TURING 图灵程序设计丛书

你不知道的JavaScript（下卷）

You Don't Know JavaScript: Up & Going, ES6 & Beyond

［美］Kyle Simpson 著
单业 译

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®
O'Reilly Media, Inc.授权人民邮电出版社出版

人民邮电出版社
北　　京

图书在版编目（C I P）数据

你不知道的JavaScript. 下卷 / (美) 凯尔·辛普森
(Kyle Simpson) 著 ; 单业译. -- 北京 : 人民邮电出版
社, 2018.1
（图灵程序设计丛书）
ISBN 978-7-115-47165-9

Ⅰ. ①你… Ⅱ. ①凯… ②单… Ⅲ. ①JAVA语言—程
序设计 Ⅳ. ①TP312.8

中国版本图书馆CIP数据核字(2017)第268838号

内 容 提 要

JavaScript 这门语言简单易用，很容易上手，但其语言机制复杂微妙，即使是经验丰富的 JavaScript 开发人员，如果没有认真学习的话也无法真正理解。本套书直面当前 JavaScript 开发人员不求甚解的大趋势，深入理解语言内部的机制，全面介绍了 JavaScript 中常被人误解和忽视的重要知识点。本书是其下卷，主要介绍了 JavaScript 入门知识和对 ES6 及未来发展趋势的展望。

本书既适合 JavaScript 语言初学者了解其精髓，又适合经验丰富的 JavaScript 开发人员深入学习。

◆ 著　　[美] Kyle Simpson
译　　单　业
责任编辑　朱　巍
执行编辑　张海艳
责任印制　彭志环

◆ 人民邮电出版社出版发行　　北京市丰台区成寿寺路11号
邮编　100164　　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京　　　　印刷

◆ 开本：800×1000　1/16
印张：17.75
字数：420千字　　2018年 1 月第 1 版
印数：1－4 000册　　2018年 1 月北京第 1 次印刷

著作权合同登记号　图字：01-2016-4640号

定价：79.00元

读者服务热线：(010)51095186转600　印装质量热线：(010)81055316

反盗版热线：(010)81055315

广告经营许可证：京东工商广登字 20170147 号

版权声明

O’Reilly Media, Inc.介绍

O’Reilly Media 通过图书、杂志、在线服务、调查研究和会议等方式传播创新知识。自 1978 年开始，O’Reilly 一直都是前沿发展的见证者和推动者。超级极客们正在开创着未来，而我们关注真正重要的技术趋势——通过放大那些“细微的信号”来刺激社会对新科技的应用。作为技术社区中活跃的参与者，O’Reilly 的发展充满了对创新的倡导、创造和发扬光大。

O’Reilly 为软件开发人员带来革命性的“动物书”；创建第一个商业网站（GNN）；组织了影响深远的开放源代码峰会，以至于开源软件运动以此命名；创立了 *Make* 杂志，从而成为 DIY 革命的主要先锋；公司一如既往地通过多种形式缔结信息与人的纽带。O’Reilly 的会议和峰会集聚了众多超级极客和高瞻远瞩的商业领袖，共同描绘出开创新产业的革命性思想。作为技术人士获取信息的选择，O’Reilly 现在还将先锋专家的知识传递给普通的计算机用户。无论是通过书籍出版、在线服务或者面授课程，每一项 O’Reilly 的产品都反映了公司不可动摇的理念——信息是激发创新的力量。

业界评论

“O’Reilly Radar 博客有口皆碑。”

——*Wired*

“O’Reilly 凭借一系列（真希望当初我也想到了）非凡想法建立了数百万美元的业务。”

——*Business 2.0*

“O’Reilly Conference 是聚集关键思想领袖的绝对典范。”

——*CRN*

“一本 O’Reilly 的书就代表一个有用、有前途、需要学习的主题。”

——*Irish Times*

“Tim 是位特立独行的商人，他不光放眼于最长远、最广阔的视野，并且切实地按照 Yogi Berra 的建议去做了：‘如果你在路上遇到岔路口，走小路（岔路）。’回顾过去，Tim 似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。”

——*Linux Journal*

目录

第二部分　ES6 及更新版本

前言

我相信你已经注意到了这一系列图书的封面上都有大大的“JS”，它并不是用来诅咒JavaScript的缩写，尽管我们大家都诅咒过这门语言的怪异之处。

从最早期的Web开始，JavaScript就是驱动内容消费的交互式体验的基本技术。尽管闪烁的鼠标轨迹和恼人的弹出式广告可能是JavaScript起步的地方。但是近二十年之后，JavaScript的技术和功能已经有了很大的发展，并且位于世界上使用最广泛的软件平台——Web的核心，它的重要性几乎没有人再会质疑。

但是，作为一门编程语言，JavaScript一直为人诟病，部分原因是其历史沿革，更重要的原因则是其设计理念。因为JavaScript这个名字，Brendan Eich曾戏称它为“傻小弟”（相对于成熟的Java而言）。实际上，这个名字完全是政治和市场考量下的产物。两门语言之间千差万别，“JavaScript”之于“Java”就如同“Carnival”（嘉年华）之于“Car”（汽车）一样，两者之间并无半点关系。

JavaScript在概念和语法风格上借鉴了其他编程语言，包括C风格的过程式编程和隐晦的Scheme/Lisp风格的函数式编程，这使得它能为不同背景的开发人员所接受，包括那些没有多少编程经验的人。用JavaScript编写一个“Hello World”程序非常简单。因此对于初学者而言，它是有吸引力和易学的。

JavaScript可能是最容易上手的编程语言之一，但它的一些奇特之处使得它不像其他语言那样容易完全掌握。要想用C或者C++开发一个完整的应用程序，开发者需要对该门语言有相当深入的了解。然而对于JavaScript，即使我们用它开发了一个完整的系统也不见得就能深入理解它。

这门语言中有些复杂的概念隐藏得很深，却常常以一种看似简单的形式呈现。例如，将函数作为回调函数传递，这让JavaScript开发人员往往满足于使用这些现成便利的机制，而不愿去探究其中的原理。

JavaScript 是一门简单易用的语言，应用广泛，同时它的语言机制又十分复杂和微妙，即使经验丰富的开发人员也需要用心学习才能真正掌握。

JavaScript 的矛盾之处就在于此，它的阿喀琉斯之踵正是本书要解决的问题。因为无需深入理解就能用它来编程，所以人们常常放松对它的学习。

使命

在学习 JavaScript 的过程中，碰到令人抓狂的问题或挫折时，如果置之不理或不求甚解（就像有些人习惯做的那样），我们很快就会发现自己根本无从发挥这门语言的威力。

尽管这些被称为 JavaScript 的“精华”部分，但我恳请读者朋友们将其看作“容易的”“安全的”或者“不完整的”部分。

“你不知道的 JavaScript”系列丛书旨在介绍 JavaScript 的另一面，让你深入掌握 JavaScript 的全部，特别是那些难点。

JavaScript 开发人员常常满足于一知半解，不愿更深入地了解其深层原因和运作方式，本书要解决的正是这个问题。我们会直面那些疑难困惑，绝不回避。

我个人不会仅仅满足于让代码运行起来而不明就里，你也应该这样。本书中，我会逐步介绍 JavaScript 中那些不太为人所知的地方，最终让你对这门语言有一个全面的了解。一旦掌握了这些知识，那些技巧、框架和时髦术语等都将不在话下。

本系列丛书全面深入地介绍了 JavaScript 中常为人误解和忽视的重要知识点，让你在读完之后不论从理论上还是实践上都能对这门语言有足够的信心。

目前你对 JavaScript 的了解可能都来自那些自身就一知半解的“专家”，而这仅仅是冰山一角。读完本系列丛书后，你将真正了解这门语言。现在就让我们踏上阅读寻知之旅吧。

综述

JavaScript 是一门优秀的语言。只学其中一部分内容很容易，但是要全面掌握则很难。开发人员遇到困难时往往将其归咎于语言本身，而不反省他们自己对语言的理解有多匮乏。本系列丛书旨在解决这个问题，使读者能够发自内心地喜欢上这门语言。



本书中的很多示例都假定你使用的是现代（以及未来）的 JavaScript 引擎环境，比如 ES6。有些代码在旧版本（ES6 之前）的引擎下可能不会像书中描述的那样工作。

排版约定

本书使用了下列排版约定。

- **黑体**

  表示新术语或重点强调的内容。

- 等宽字体（`constant width`）

  表示程序片段，以及正文中出现的变量、函数名、数据库、数据类型、环境变量、语句和关键字等。

- 加粗等宽字体（**`constant width bold`**）

  表示应该由用户输入的命令或其他文本。

- 等宽斜体（*`constant width italic`*）

  表示应该由用户输入的值或根据上下文确定的值替换的文本。

该图标表示提示或建议。

该图标表示一般注记。

该图标表示警告或警示。

使用代码示例

补充材料（代码示例、练习等）可以从 http://bit.ly/ydkjs-up-going-code 和 http://bit.ly/ydkjs-es6beyond-code 下载。

本书是要帮你完成工作的。一般来说，如果本书提供了示例代码，你可以把它用在你的程序或文档中。除非你使用了很大一部分代码，否则无需联系我们获得许可。比如，用本书的几个代码片段写一个程序就无需获得许可，销售或分发 O'Reilly 图书的示例光盘则需要获得许可；引用本书中的示例代码回答问题无需获得许可，将书中大量的代码放到你的产

品文档中则需要获得许可。

我们很希望但并不强制要求你在引用书中内容时加上引用说明。引用说明一般包括书名、作者、出版社和ISBN。比如："*You Don't Know JavaScript: Up & Going* by Kyle Simpson (O'Reilly). Copyright 2015 Getify Solutions, Inc., 978-1-491-92446-4"。

如果你觉得自己对示例代码的用法超出了上述许可的范围，欢迎你通过 permissions@oreilly.com 与我们联系。

Safari® Books Online

Safari Books Online（http://www.safaribooksonline.com）是应运而生的数字图书馆。它同时以图书和视频的形式出版世界顶级技术和商务作家的专业作品。

技术专家、软件开发人员、Web 设计师、商务人士和创意专家等，在开展调研、解决问题、学习和认证培训时，都将 Safari Books Online 视作获取资料的首选渠道。

对于组织团体、政府机构和个人，Safari Books Online 提供各种产品组合和灵活的定价策略。用户可通过一个功能完备的数据库检索系统访问 O'Reilly Media、Prentice Hall Professional、Addison-Wesley Professional、Microsoft Press、Sams、Que、Peachpit Press、Focal Press、Cisco Press、John Wiley & Sons、Syngress、Morgan Kaufmann、IBM Redbooks、Packt、Adobe Press、FT Press、Apress、Manning、New Riders、McGraw-Hill、Jones & Bartlett、Course Technology 以及其他几十家出版社的上千种图书、培训视频和正式出版之前的书稿。要了解 Safari Books Online 的更多信息，我们网上见。

联系我们

请把对本书的评价和问题发给出版社。

美国：

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472

中国：

北京市西城区西直门南大街 2 号成铭大厦 C 座 807 室（100035）
奥莱利技术咨询（北京）有限公司

O'Reilly 的每一本书都有专属网页，你可以在那儿找到本书的相关信息，包括勘误表、示

例代码以及其他信息。本书第一部分“起步上路”的网站地址是 http://bit.ly/ydkjs_up-and-going。本书第二部分“ES6 及更新版本”的网站地址是：http://bit.ly/ydkjs-es6-beyond。

对于本书的评论和技术性问题，请发送电子邮件到：bookquestions@oreilly.com

要了解更多 O’Reilly 图书、培训课程、会议和新闻的信息，请访问以下网站：

  http://www.oreilly.com

我们在 Facebook 的地址如下：http://facebook.com/oreilly

请关注我们的 Twitter 动态：http://twitter.com/oreillymedia

我们的 YouTube 视频地址如下：http://www.youtube.com/oreillymedia

电子书

扫描如下二维码，即可购买本书电子版。

致谢

我要感谢很多人，是他们的帮助让本书以及整个系列得以出版。

首先，我必须感谢我的妻子 Christen Simpson 以及我的两个孩子 Ethan 和 Emily，容忍我整天坐在电脑前工作。即使不写作的时候，我的眼睛也总是盯着屏幕做一些与 JavaScript 相关的工作。我牺牲了很多陪伴家人的时间，这个系列的丛书才得以为读者深入全面地介绍 JavaScript。对于家庭，我亏欠太多。

我要感谢 O'Reilly 的编辑 Simon St.Laurent 和 Brian MacDonald，以及所有其他的编辑和市场工作人员。和他们一起工作非常愉快；本系列丛书的写作、编辑和制作都以开源方式进行，在此实验过程中，他们给予了非常多的帮助。

我要感谢所有为本系列丛书提供建议和校正的人，包括 Shelley Powers、Tim Ferro、Evan Borden、Forrest L. Norvell、Jennifer Davis、Jesse Harlin 等。十分感谢 Jenn Lukas 和 Rick Waldron 为本书作序。

我要感谢 JavaScript 社区中的许多人，包括 TC39 委员会的成员们，将他们的知识与我们分享，并且耐心详尽地回答我无休止的提问。他们是 John-David Dalton、Juriy "kangax" Zaytsev、Mathias Bynens、Axel Rauschmayer、Nicholas Zakas、Angus Croll、Reginald Braithwaite、Dave Herman、Brendan Eich、Allen Wirfs-Brock、Bradley Meck、Domenic Denicola、David Walsh、Tim Disney、Peter van der Zee、Andrea Giammarchi、Kit Cambridge、Eric Elliott、André Bargull、Caitlin Potter、Brian Terlson、Ingvar Stepanyan、Chris Dickinson、Luke Hoban，等等。还有很多人，我无法一一感谢。

"你不知道的 JavaScript"系列丛书是由 Kickstarter 发起的，我要感谢近 500 名慷慨的支持者，没有他们的支持就没有这套系列丛书：

Jan Szpila、nokiko、Murali Krishnamoorthy、Ryan Joy、Craig Patchett、pdqtrader、Dale Fukami、ray hatfield、R0drigo Perez [Mx]、Dan Petitt、Jack Franklin、Andrew Berry、Brian

Grinstead、Rob Sutherland、Sergi Meseguer、Phillip Gourley、Mark Watson、Jeff Carouth、Alfredo Sumaran、Martin Sachse、Marcio Barrios、Dan、AimelyneM、Matt Sullivan、Delnatte Pierre-Antoin、Jake Smith、Eugen Tudorancea、Iris、David Trinh、simonstl、Ray Daly、Uros Gruber、Justin Myers、Shai Zonis、Mom & Dad、Devin Clark、Dennis Palmer、Brian Panahi Johnson、Josh Marshall、Marshall、Dennis Kerr、Matt Steele、Erik Slagter、Sacah、Justin Rainbow、Christian Nilsson、Delapouite、D.Pereira、Nicolas Hoizey、George V. Reilly、Dan Reeves、Bruno Laturner、Chad Jennings、Shane King、Jeremiah Lee Cohick、od3n、Stan Yamane、Marko Vucinic、Jim B、Stephen Collins、Egir Porsteinsson、Eric Pederson、Owain、Nathan Smith、Jeanetteurphy、Alexandre ELISé、Chris Peterson、Rik Watson、Luke Matthews、Justin Lowery、Morten Nielsen、Vernon Kesner、Chetan Shenoy、Paul Tregoing、Marc Grabanski、Dion Almaer、Andrew Sullivan、Keith Elsass、Tom Burke、Brian Ashenfelter、David Stuart、Karl Swedberg、Graeme、Brandon Hays、John Christopher、Gior、manoj reddy、Chad Smith、Jared Harbour、Minoru TODA、Chris Wigley、Daniel Mee、Mike、Handyface、Alex Jahraus、Carl Furrow、Rob Foulkrod、Max Shishkin、Leigh Penny Jr.、Robert Ferguson、Mike van Hoenselaar、Hasse Schougaard、rajan venkataguru、Jeff Adams、Trae Robbins、Rolf Langenhuijzen、Jorge Antunes、Alex Koloskov、Hugh Greenish、Tim Jones、Jose Ochoa、Michael Brennan-White、Naga Harish Muvva、Barkóczi Dávid、Kitt Hodsden、Paul McGraw、Sascha Goldhofer、Andrew Metcalf、Markus Krogh、Michael Mathews、Matt Jared、Juanfran、Georgie Kirschner、Kenny Lee、Ted Zhang、Amit Pahwa、Inbal Sinai、Dan Raine、Schabse Laks、Michael Tervoort、Alexandre Abreu、Alan Joseph Williams、NicolasD、Cindy Wong、Reg Braithwaite、LocalPCGuy、Jon Friskics、Chris Merriman、John Pena、Jacob Katz、Sue Lockwood、Magnus Johansson、Jeremy Crapsey、Grzegorz Pawlowski、nico nuzzaci、Christine Wilks、Hans Bergren、charles montgomery、Ariel-Fogel、Ivan Kolev、Daniel Campos、Hugh Wood、Christian Bradford、Frédéric Harper、Ionut Dan Popa、Jeff Trimble、Rupert Wood、Trey Carrico、Pancho Lopez、Joel kuijten、Tom A Marra、Jeff Jewiss、Jacob Rios、Paolo Di Stefano、Soledad Penades、Chris Gerber、Andrey Dolganov、Wil Moore III、Thomas Martineau、Kareem、Ben Thouret、Udi Nir、Morgan Laupies、jory carson-burson、Nathan L Smith、Eric Damon Walters、Derry Lozano-Hoyland、Geoffrey Wiseman、mkeehner、KatieK、Scott MacFarlane、Brian LaShomb、Adrien Mas、christopher ross、Ian Littman、Dan Atkinson、Elliot Jobe、Nick Dozier、Peter Wooley、John Hoover、dan、Martin A. Jackson、Héctor Fernando Hurtado、andy ennamorato、Paul Seltmann、Melissa Gore、Dave Pollard、Jack Smith、Philip Da Silva、Guy Israeli、@megalithic、Damian Crawford、Felix Gliesche、April Carter Grant、Heidi、jim tierney、Andrea Giammarchi、Nico Vignola、Don Jones、Chris Hartjes、Alex Howes、john gibbon、David J. Groom、BBox、Yu Dilys Sun、Nate Steiner、Brandon Satrom、Brian Wyant、Wesley Hales、Ian Pouncey、Timothy Kevin Oxley、George Terezakis、sanjay raj、Jordan Harband、Marko McLion、Wolfgang Kaufmann、Pascal

Peuckert、Dave Nugent、Markus Liebelt、Welling Guzman、Nick Cooley、Daniel Mesquita、Robert Syvarth、Chris Coyier、Rémy Bach、Adam Dougal、Alistair Duggin、David Loidolt、Ed Richer、Brian Chenault、GoldFire Studios、Carles Andrés、Carlos Cabo、Yuya Saito、roberto ricardo、Barnett Klane、Mike Moore、Kevin Marx、Justin Love、Joe Taylor、Paul Dijou、Michael Kohler、Rob Cassie、Mike Tierney、Cody Leroy Lindley、tofuji、Shimon Schwartz、Raymond、Luc De Brouwer、David Hayes、Rhys Brett-Bowen、Dmitry、Aziz Khoury、Dean、Scott Tolinski-Level Up、Clement Boirie、Djordje Lukic、Anton Kotenko、Rafael Corral、Philip Hurwitz、Jonathan Pidgeon、Jason Campbell、Joseph C.、SwiftOne、Jan Hohner、Derick Bailey、getify、Daniel Cousineau、Chris Charlton、Eric Turner、David Turner、Joel Galeran、Dharma Vagabond、adam、Dirk van Bergen、dave ♥♫★ furf、Vedran Zakanj、Ryan McAllen、Natalie Patrice Tucker、Eric J. Bivona、Adam Spooner、Aaron Cavano、Kelly Packer、Eric J、Martin Drenovac、Emilis、Michael Pelikan、Scott F. Walter、Josh Freeman、Brandon Hudgeons、vijay chennupati、Bill Glennon、Robin R.、Troy Forster、otaku_coder、Brad、Scott、Frederick Ostrander、Adam Brill、Seb Flippence、Michael Anderson、Jacob、Adam Randlett、Standard、Joshua Clanton、Sebastian Kouba、Chris Deck、SwordFire、Hannes Papenberg、Richard Woeber、hnzz、Rob Crowther、Jedidiah Broadbent、Sergey Chernyshev、Jay-Ar Jamon、Ben Combee、luciano bonachela、Mark Tomlinson、Kit Cambridge、Michael Melgares、Jacob Adams、Adrian Bruinhout、Bev Wieber、Scott Puleo、Thomas Herzog、April Leone、Daniel Mizieliński、Kees van Ginkel、Jon Abrams、Erwin Heiser、Avi Laviad、David newell、Jean-Francois Turcot、Niko Roberts、Erik Dana、Charles Neill、Aaron Holmes、Grzegorz Ziólkowski、Nathan Youngman、Timothy、Jacob Mather、Michael Allan、Mohit Seth、Ryan Ewing、Benjamin Van Treese、Marcelo Santos、Denis Wolf、Phil Keys、Chris Yung、Timo Tijhof、Martin Lekvall、Agendine、Greg Whitworth、Helen Humphrey、Dougal Campbell、Johannes Harth、Bruno Girin、Brian Hough、Darren Newton、Craig McPheat、Olivier Tille、Dennis Roethig、Mathias Bynens、Brendan Stromberger、sundeep、John Meyer、Ron Male、John F Croston III、gigante、Carl Bergenhem、B.J. May、Rebekah Tyler、Ted Foxberry、Jordan Reese、Terry Suitor、afeliz、Tom Kiefer、Darragh Duffy、Kevin Vanderbeken、Andy Pearson、Simon Mac Donald、Abid Din、Chris Joel、Tomas Theunissen、David Dick、Paul Grock、Brandon Wood、John Weis、dgrebb、Nick Jenkins、Chuck Lane、Johnny Megahan、marzsman、Tatu Tamminen、Geoffrey Knauth、Alexander Tarmolov、Jeremy Tymes、Chad Auld、Sean Parmelee、Rob Staenke、Dan Bender、Yannick derwa、Joshua Jones、Geert Plaisier、Tom LeZotte、Christen Simpson、Stefan Bruvik、Justin Falcone、Carlos Santana、Michael Weiss、Pablo Villoslada、Peter deHaan、Dimitris Iliopoulos、seyDoggy、Adam Jordens、Noah Kantrowitz、Amol M、Matthew Winnard、Dirk Ginader、Phinam Bui、David Rapson、Andrew Baxter、Florian Bougel、Michael George、Alban Escalier、Daniel Sellers、Sasha Rudan、John Green、Robert Kowalski、David I. Teixeira (@ditma)、Charles Carpenter、

Justin Yost、Sam S、Denis Ciccale、Kevin Sheurs、Yannick Croissant、Pau Fracés、Stephen McGowan、Shawn Searcy、Chris Ruppel、Kevin Lamping、Jessica Campbell、Christopher Schmitt、Sablons、Jonathan Reisdorf、Bunni Gek、Teddy Huff、Michael Mullany、Michael Fürstenberg、Carl Henderson、Rick Yoesting、Scott Nichols、Hernán Ciudad、Andrew Maier、Mike Stapp、Jesse Shawl、Sérgio Lopes、jsulak、Shawn Price、Joel Clermont、Chris Ridmann、Sean Timm、Jason Finch、Aiden Montgomery、Elijah Manor、Derek Gathright、Jesse Harlin、Dillon Curry、Courtney Myers、Diego Cadenas、Arne de Bree、Joao Paulo Dubas、James Taylor、Philipp Kraeutli、Mihai Paun、Sam Gharegozlou、joshjs、Matt Murchison、Eric Windham、Timo Behrmann、Andrew Hall、joshua price、Théophile Villard

这套系列丛书的写作、编辑和制作都是以开源的方式进行的。我要感谢 GitHub 让这一切成为可能！

再次向我没能提及的支持者们表示感谢。这套系列丛书属于我们每一个人，希望它能够帮助更多的人更好地了解 JavaScript，让当前和未来的社区贡献者受益。

第一部分

起步上路

序

你最近新学的技能是什么？

可能是一门外语，比如意大利语或德语。可能是一个图像编辑工具，如 Photoshop。也可能是某种厨艺或者木工活，又或是某种健身项目。请回忆一下你最终掌握这项技能的那个时刻，那应该就是你突然顿悟的一刻。当你学会操控台锯或者理解了法语中阳性名词和阴性名词之间的区别时，事情就从模糊变得明朗清晰起来。那时感觉如何？是不是非常不可思议？

现在，再往前回忆一下你掌握这项新技能之前的情形。那时你是什么感觉呢？可能有点恐惧又有点心慌，是不是？在某一刻，我们还不了解自己现在已经掌握的知识，这完全没有任何问题；每个人都是从某个起点开始学习的。学习新技能是一场令人激动的探险，特别是当你想要高效学习某个主题时。

我教授过很多初级的编程课程。跟着我学习的学生通常之前已经试着通过博客或复制、粘贴代码来自学过 HTML 或者 JavaScript 这样的主题，但是他们并没有能够真正掌握编码知识来实现想要的目标。而且，由于并没有真正掌握某些编程主题的细节，他们便无法编写功能强大的代码或调试自己的作品，因为他们并没有真正理解所发生的一切。

我一直坚信要以正确的方法授课，这意味着我会讲授 Web 标准、语义标记、注释良好的代码以及其他的最佳实践。我会详细讲解涉及的主题，解释如何做以及这么做的原因，而不仅仅是扔出代码以供复制、粘贴。努力理解自己的代码后，你就可以更好地完成任务，同时自己也会得到进步。此时代码不再仅仅只是你的**工作**，更是你的**作品**。这就是我喜欢本书内容的原因。Kyle 带领我们深入了解语法和术语，对 JavaScript 这个语言进行了出色、全面的介绍。本书并没有流于表面，而是确实有助于我们真正理解概念。

就像只学会如何在 Photoshop 中打开、关闭和保存文档是不够的，只能够将 jQuery 代码片段复制到你的网站上也是不够的。的确，只需要学习一些与编程相关的基础知识，我就能

够编写并共享自己的设计。但如果没有对工具及其背后原理的正确理解，我怎么能够定义一个网格或创建一个明晰的类型系统呢？又如何才能优化 Web 上的图像呢？对 JavaScript 来说也是一样的。如果不清楚循环的工作模式、变量的定义以及作用域的含义，那么我们就无法编写自己所能实现的最佳代码。我们不能接受退而求其次的作品，毕竟这是我们自己的作品。

你对 JavaScript 的探索越深入，它就会变得越清晰。或许你对闭包、对象和方法这样的词汇目前来说还不是很熟悉，但本书将帮助你明晰这些术语。我希望你在开始学习本书时记住自己学习某样东西前后的感受。这可能很艰巨，但为了要开始一段磨砺你的知识宝剑的精彩之旅，你已经翻阅至此。本书第一部分是我们理解编程的起点。享受你顿悟的时刻吧！

——Jenn Lukas（http://jennlukas.com, @jennlukas），前端顾问

第 1 章

深入编程

欢迎来到“你不知道的 JavaScript”系列。

本部分介绍了编程中的一系列基本概念，并有助于你更好地理解本系列其余几本书的内容。当然，本部分的内容会更偏向于 JavaScript（常简写为 JS）。如果你是刚开始学习编程或 JavaScript，那么本部分将会简单探讨你起步和继续学习所需要了解的概念。

本部分一开始会从很高的层次来介绍编程的基本原则。基本上假设你在阅读“你不知道的 JavaScript”系列图书时几乎没有编程经历，并想要通过学习这几本书从 JavaScript 这个视角开始理解编程。

第 1 章总结了深入学习和实践**编程**所需要的知识，此外还有许多其他介绍编程的优秀资源，这些资源可以帮助你更深入地探索这些主题。除了第 1 章的知识，我建议你还要利用这些资料来更深入地学习。

在熟悉了常用的基础编程概念后，第 2 章将帮助你熟悉 JavaScript 的编程风格。第 2 章介绍了 JavaScript 的含义，但再次声明，这并不是完整的指南——这是“你不知道的 JavaScript”系列其余几本图书的主旨！

如果你已经对 JavaScript 有了一定的了解，那么可以先查看第 3 章来了解“你不知道的 JavaScript”系列图书的简介，然后直接阅读你所感兴趣的章节！

1.1 代码

让我们从头开始。

程序常被称为**源码**或**代码**，它是一组特定的指令，用来指示计算机要执行哪些任务。虽然对 JavaScript 来说可以直接在浏览器的开发者终端中输入代码，但代码通常会被保存在文本文件中，我们将在后文中对此进行简单介绍。

指令的格式和组合规则被称为**计算机语言**，有时也被称为**语法**，这非常类似于英语中告诉你如何拼写单词以及如何使用单词和标点符号来构造有效的句子。

语句

在计算机语言中，执行特定任务的一组单词、数字和运算符被称为**语句**。在 JavaScript 中，一条语句可能如下所示：

```
a = b * 2;
```

其中的字符 `a` 和 `b` 称为**变量**（参见 1.7 节），它们就好比是可以存放东西的小盒子。在程序中，变量保存程序要使用的值（比如数字 `42`）。你可以将它们想象成值本身的替代符。

相比之下，`2` 本身就是一个值，称为**字面值**，因为它独立存在而没有保存在变量之中。

其中的字符 `=` 和 `*` 是**运算符**（参见 1.4 节），它们对值和变量执行动作，如赋值和进行乘法运算。

JavaScript 的多数语句都是以分号（`;`）结尾的。

粗略地说，语句 `a = b * 2;` 告诉计算机获取变量 `b` 的当前值，然后将这个值乘以 `2`，再将计算结果保存到另一个名为 `a` 的变量中。

程序就是多个这样语句的集合，它们合起来描述了程序要执行的所有步骤。

1.2　表达式

语句由一个或多个**表达式**组成。一个表达式是对一个变量或值的引用，或者是一组值和变量与运算符的组合。

举例来说，`a = b * 2;` 这个语句中有四个表达式。

- `2` 是一个**字面值表达式**。
- `b` 是一个**变量表达式**，表示获取它的当前值。
- `b * 2` 是一个**算术表达式**，表示进行乘法运算。
- `a = b * 2` 是一个**赋值表达式**，意思是将表达式 `b * 2` 的结果赋值给变量 `a`（我们将在后文中深入介绍赋值）。

一个独立的表达式也可以称为**表达式语句**，如下所示：

```
b * 2;
```

这种表达式语句不是很常用，或者说不是很有用，因为它通常不会对程序的运行起到任何作用，它只是取得 b 的值并乘以 2，但是却没有对结果有任何影响。

更常用的表达式语句是**调用表达式**语句（参见 1.11 节），因为整个语句本身就是一个函数调用表达式：

```
alert( a );
```

执行程序

这些编程语句的集合是如何通知计算机来执行任务的呢？程序需要**被执行**，我们也将这一过程称为**运行程序**。

a = b * 2 这样的语句便于开发者读写，但实际上计算机并不能直接理解这种形式。因此，需要通过计算机上一个专门的工具（**解释器**或**编译器**）将你编写的代码翻译成计算机可以理解的命令。

对某些计算机语言来说，在程序被执行时，对命令的翻译通常是自上而下逐行执行的，这通常被称为**代码解释**。

对另外一些语言来说，这种翻译是预先进行的，被称为**代码编译**，这样一来，当**执行**程序时，实际上运行的是已经编译好的、可以执行的计算机指令。

基本上可以说 JavaScript 是**解释型的**，因为每次执行 JavaScript 源码时都需要进行处理。但这么说并不完全精确。JavaScript 引擎实际上是**动态编译**程序，然后立即执行编译后的代码。

> 有关 JavaScript 编译的更多信息，参见本系列《你不知道的 JavaScript（上卷）》[1] 第一部分中的前两章。

1.3　实践

本章将通过简单的代码片段来介绍每个编程概念，这些代码（当然）是用 JavaScript 编写的。

非常重要的一点是，在阅读本章时，你应该通过亲自编写代码来实践每个概念，并且你可能需要花一点时间反复阅读本章。最简单的方法是，使用最方便的浏览器（Firefox、Chrome、IE 等）的开发者工具来实践。

注 1：此书已由人民邮电出版社出版。——编者注

一般来说，你可以通过菜单项或者快捷键来打开开发者终端。有关在你喜欢的浏览器中打开和使用终端的更多详细信息，参见“掌握开发者工具终端”（http://blog.teamtreehouse.com/mastering-developer-tools-console）。

如果要在终端中一次输入多行，那么可以使用 <shift>+<enter> 组合键来另起一行。一旦点击 <enter> 键，终端会立即执行已输入的所有代码。

我们来熟悉一下在终端中运行代码的流程。首先，建议你在浏览器中打开一个空白的标签页。我更喜欢在地址栏中输入 about:blank 来实现这一点。然后，确保你的开发者终端是开启状态，就像我们之前提到的那样。

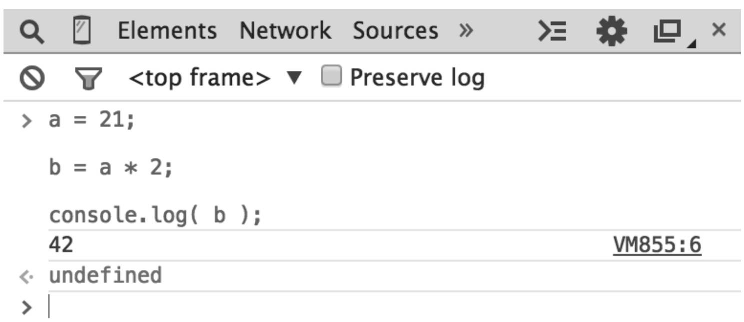
现在，输入如下代码，并观察代码的执行：

```
a = 21;

b = a * 2;

console.log( b );
```

在 Chrome 浏览器的终端中输入前面的代码将会产生如下所示的输出：



你可以自己试一下。学习编程的最好方法就是编写代码！

1.3.1 输出

在前面的代码片段中，我们使用了 console.log(..)。现在我们来简单了解一下这行代码做了些什么。

可能你已经猜到了，这就是我们在开发者终端打印文本（即向用户**输出**）的方法。我们应该对这个语句的两点解释一下。

首先，log(b) 这一部分是一个函数调用（参见 1.11 节）。我们将变量 b 传给这个函数，请求它将 b 的值打印到终端中。

其次，`console.` 这一部分是 `log(..)` 函数所在的对象引用。我们将在第 2 章中深入介绍对象及其属性。

创建可见输出的另外一个方法是运行 `alert(..)` 语句。如下所示：

```
alert( b );
```

如果运行这个语句，那么你就会发现输出并没有打印到终端中，而是弹出一个“OK”对话框，变量 `b` 的内容会呈现在对话框中。然而，在终端中学习和运行程序时，使用 `console.log(..)` 通常比使用 `alert(..)` 更加方便，因为这样无需与浏览器界面交互就可以一次输出多个变量。

我们在本部分中使用 `console.log(..)` 作为输出方法。

1.3.2　输入

在讨论输出的同时，你可能也会好奇如何实现**输入**（即如何接收用户的信息）。

最常用的方法是，通过 HTML 页面向用户显示表单元素（如文本框）用于输入，然后通过 JavaScript 将这些值读取到程序变量中。

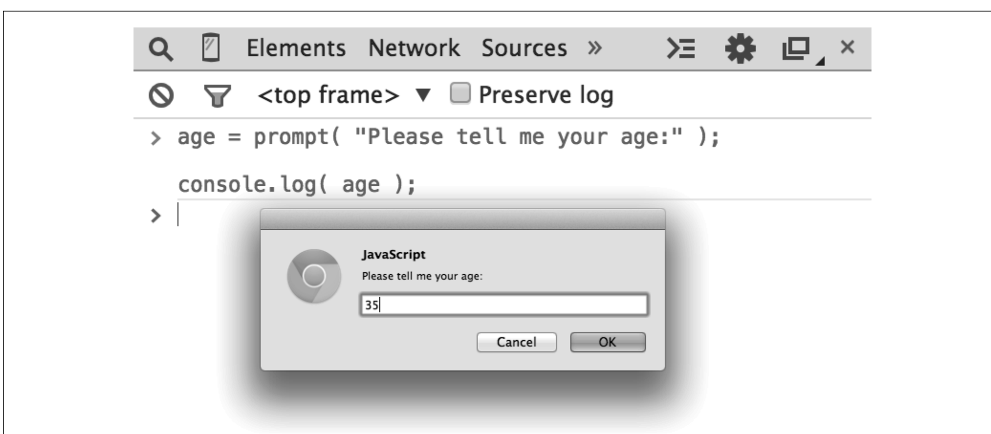
还有另一种更为简单的获取输入的方法，用于简单的学习和展示，这也是我们将会使用的方法，即 `prompt(..)` 函数：

```
age = prompt( "Please tell me your age:" );

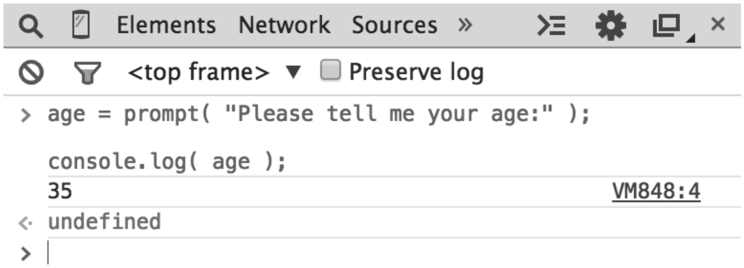
console.log( age );
```

你可能已经猜到了，传给 `prompt(..)` 的消息会打印到弹出窗口中，本例中是 `"Please tell me your age:"`。

如下图所示：



输入文本并点击“OK”后，你就可以看到输入的值会保存到变量 age 中，接着通过 console.log(..) 输出：



为了简化难度，在学习基本编程概念时，本部分中使用的示例都不需要输入。但既然你已经学习了如何使用 prompt(..)，如果想要挑战自我，你可以在自己的示例中使用输入。

1.4 运算符

使用运算符，我们可以对变量和值执行操作。我们已经在前文中看到了 = 和 * 这两个 JavaScrit 运算符。

运算符 * 执行算术乘法。很简单，对吧？

等号运算符 = 用于**赋值**——我们先计算 = **右边**（源值）的值，然后将它存入**左边**（目标变量）指定的变量中。



这种赋值方法的顺序看起来是反的，有点奇怪。有些人可能更习惯将顺序调过来，将源值放在左边，目标变量放在右边，不用 a = 42 这种形式，而是 42 -> a（这不是合法的 JavaScript）。但问题是，a = 42 这种顺序以及类似的变体在现代编程语言中是非常流行的。如果感觉不太习惯的话，那么你就要花点时间来习惯它，并将它植入到你的思维中。

考虑：

```
a = 2;
b = a + 1;
```

在上述示例中，我们将值 2 赋给变量 a。然后，我们取得变量 a 的值（仍然是 2），加上 1，得到结果 3，再将这个值保存在变量 b 中。

虽然 var 严格意义上说并不是一个运算符，但每个程序都会用到这个关键词，因为它是**声明**（也就是**创建**）变量（参见 1.7 节）的基本方法。

在使用变量前总是应该先声明变量。一个变量在每个**作用域**（参见 1.11 节）中只需要声明一次；声明之后可以按照需要多次使用。如下所示：

```
var a = 20;

a = a + 1;
a = a * 2;
console.log( a );  // 42
```

以下是 JavaScript 中最常用的一些运算符。

- 赋值
  =，如 a = 2 就表示将值 2 保存在变量 a 中。

- 算术
  +（加）、-（减）、*（乘）、/（除），如 a * 3。

- 复合赋值
  +=、-=、*= 和 /= 是复合运算符，可以将算术运算符与赋值组合起来，比如，a += 2 等同于 a = a + 2。

- 递增 / 递减
  ++ 表示递增，-- 表示递减，比如 a++ 就类似于 a = a + 1。

- 对象属性访问
  如 console.log() 中的 .。

  对象是在名为属性的位置中持有其他值的值。obj.a 指的是一个名为 obj 的对象值，并伴有一个属性名为 a 的属性。也可以通过 obj["a"] 这种形式访问属性。参见第 2 章。

- 相等
  ==（粗略相等）、===（严格相等）、!=（粗略不等）和 !==（严格不等），如 a == b。

  参见 2.1 节。

- 比较
  <（小于）、>（大于）、<=（小于或粗略等于）和 >=（大于或粗略等于），如 a <= b。

  参见 2.1 节。

- 逻辑
  &&（与）和 ||（或），如 a || b 就表示 a 或者 b。

  这些运算符用于表示复合条件（参见 1.9 节），比如 a 或 b 为真。

有关运算符的更多细节以及这里没有覆盖到的更多介绍，参见 Mozilla 开发者网络的“表达式与运算符”（https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Expressions_and_Operators）。

1.5 值与类型

如果你向手机店的店员咨询某款手机的价格，他会回答说“99.99”（也就是 99.99 美元），那么他给你的是一个实际的美元数，表示购买手机需要付的金额（加上税）。如果你想要买两部这款手机，那么可以很容易地算出价格为 199.98 美元。

如果这个店员拿起另一个类似的手机，声称它是“免费的”（很可能要签约），这时他并没有给出一个具体的数字，而是价格（$0.00）的另外一种表示方法——“免费”。

接着，如果你询问手机是否附带充电器，那么你得到的答案只会是“是”或“否”。

同样，当在程序中表达某些值时，根据将对这些值进行的操作，你可以为这些值选择不同的表示方法。

在编程术语中，对值的不同表示方法称为**类型**。JavaScript 为以下这些**基本值**提供了内置类型。

- 在计算时，你需要的是一个数字（`number`）。
- 在屏幕上打印一个值时，你需要的是一个字符串（`string`，一个或多个字符、单词或句子）。
- 在程序中作出决策时，你需要的是一个布尔值（`boolean`，`true` 或者 `false`）。

直接包含在源码中的值被称为**字面值**。字符串字面值由双引号（`"..."`）或单引号（`'…'`）围住，二者的唯一区别只是风格不同。数字和布尔型字面值可以直接表示（如 `42`、`true` 等）。

考虑：

```
"I am a string";
'I am also a string';

42;

true;
false;
```

除了字符串 / 数字 / 布尔值类型，编程语言通常还会提供**数组**、**对象**、**函数**等。我们将在本章和下一章中更深入地介绍值和类型。

类型转换

如果需要在屏幕上打印出一个数字，那么就需要将这个值转化为字符串，在 JavaScript 中，

这种转化称为“类型转换”。类似地，如果向电子商务网页的表单中输入一系列数字字符，那么这都是字符串，但如果需要使用这些值进行数学计算，则需要将其转换为数字。

JavaScript 为**类型**间的强制转换提供了几种不同的机制。举例来说：

```
var a = "42";
var b = Number(a);

console.log( a );     // "42"
console.log( b );     // 42
```

如上所示，`Number(..)`（一个内置函数）的使用是一种**显式的**类型转换，可以将任意类型转换为数字类型。这应该是很直观的。

但是，如果需要进行比较的是不同类型的两个值，那么会怎么样呢？这就是一个很有争议的问题了，需要**隐式的**类型转换。

如果要比较字符串 `"99.99"` 和数字 `99.99`，多数人会认为它们是相等的，但它们其实并不完全相同，难道不是吗？它们是两种表示方法下的同一个值，属于两种不同的**类型**。你可以说它们是“粗略相等”，是这样吗？

为了帮助你处理这些常见的情形，JavaScript 有时会**隐式地**将值转换到匹配的类型。

因此，如果你使用 `==` 粗略相等运算符来判断 `"99.99" == 99.99` 是否成立，JavaScript 会将左边的 `"99.99"` 转换为等价的数字类型 `99.99`。这时比较就变成了 `99.99 == 99.99`，当然为 `true` 了。

尽管隐式类型转换的设计意图是为了提供便利，但如果你没有花时间学习其行为方式的规则的话，它也可能会产生误导。而多数 JavaScript 开发者从来没有花时间来学习这一知识，所以他们普遍感觉隐式类型转换令人迷惑，并且会让程序产生出乎意料的 bug。他们认为应该尽量避免隐式类型转换。隐式类型转换甚至被称为是语言设计中的缺陷。

然而，隐式类型转换是**可以学习的**机制，任何想要严肃对待 JavaScript 编程的人都应该学习。不仅仅是因为一旦掌握了其规则，就不会再被它迷惑，实际上这也可以提高你的程序质量！所以为此付出努力是十分值得的。

> 有关类型转换的更多信息，参见下一章和本系列《你不知道的 JavaScript（中卷)》[2] 第一部分中的第 4 章。

注 2：此书已由人民邮电出版社出版。——编者注

1.6　代码注释

手机商店的店员可能会草草记下新发布手机的特性或者其公司提供的新套餐。这些笔记只是给店员看的，而不是给顾客阅读的。然而，通过记录要向顾客提供的信息以及提供方式，这些笔记可以帮助店员提高自己的工作质量。

这里可以学到的最重要的一点是，编写代码并不只是为了给计算机看。在给计算机看的同时，代码同样要给开发者阅读。

计算机只关心机器码，也就是那些编译之后得到的二进制 0 和 1 序列。要想得到同样的 0 和 1 序列，几乎有无数种程序写法。而你选择的程序编写方案很重要，这不只是对你个人来说，对小组的其他成员，甚至对未来的你也同样很重要。

编写程序时不仅应该努力做到让程序能够正确执行，而且应该做到使代码阅读起来也是容易理解的。你可能需要花费很多精力为变量（参见 1.7 节）和函数（参见 1.11 节）选择一个好的名字。

另一个非常重要的部分是代码注释。这是程序中的文本，将其插入程序只是为了向人类解释说明代码的执行。解释器 / 编译器会忽略这些注释。

有关如何编写注释良好的代码有很多种观点；我们确实无法定义绝对的普遍标准。但是以下这些观察结论和指导原则是很有用的。

- 没有注释的代码不是最优的。
- 过多注释（比如每行一个）可能是拙劣代码的征兆。
- 代码应该解释**为什么**，而非**是什么**。如果编写的代码特别容易令人迷惑的话，那么注释也可以解释一下实现原理。

JavaScript 中的注释有两种类型：单行注释和多行注释。

考虑：

```
// 这是一个单行注释

/* 而这是
       一个多行
           注释。
               */
```

如果想要将注释放到单个语句上方或者一行的末尾，那么可以使用单行注释 //。这一行中位于 // 之后直到行尾的所有内容都会被当作注释（因此会被编译器忽略）。单行注释的内容没有限制。

考虑：

```
var a = 42;       // 42是生命的意义
```

多行注释 /* .. */ 适用于在注释中需要多行解释的情况。

以下是多行注释常用的一个场景：

```
/* 使用下面的值是因为
   可以看到它回答了
   宇宙中所有的问题 */
var a = 42;
```

多行注释也可以出现在行中的任意位置，甚至可以出现在行中间，因为 */ 会结束注释。如下所示：

```
var a = /* 任意值 */ 42;

console.log( a );    // 42
```

唯一不能出现在多行注释中的是 */，因为这会被解释为注释的结束。

开始学习编程时一定要养成注释代码的习惯。在本章后面的内容中，你会看到我使用注释来进行解释，所以你在自己的练习中也应该这么做。相信我，如果你这么做的话，每个阅读你代码的人都会感谢你的！

1.7 变量

大多数的实用程序都需要跟踪值的变化，因为程序在执行任务时会对值进行各种操作，值会不断发生变化。

在程序中实现这一点的最简单方法是将值赋给一个符号容器，这个符号容器称为**变量**，使用这个名字是因为这个容器中的值是可以变化的。

在某些编程语言中，你需要声明一个变量（容器）用于存放指定类型的值（如数字或字符串）。通过避免不想要的值转换，人们认为这种**静态类型**（也称为**类型强制**）提高了程序的正确性。

其他语言强调的是值的类型而不是变量的类型。**弱类型**（也称为**动态类型**）允许一个变量在任意时刻存放任意类型的值。这种方式允许一个变量在程序的逻辑流中的任意时刻代表任意类型的值，人们认为这样可以提高程序的灵活性。

JavaScript 采用了后一种机制——**动态类型**，这也就是说，变量可以持有任意类型值而不存在类型强制。

前面提到过，我们使用 var 语句声明一个变量。注意，声明中没有额外的**类型**信息。考虑以下这个简单的程序：

```
var amount = 99.99;

amount = amount * 2;

console.log( amount );      // 199.98

// 将amount转化为一个字符串， 并在开头添加 "$"
amount = "$" + String( amount );

console.log( amount );      // "$199.98"
```

变量 amount 开始时持有值 99.99，然后持有 amount * 2 的结果值，也就是 199.98。

第一个 console.log(..) 命令需要**隐式地**转换类型，将数字值转换为字符串用于输出。

然后语句 amount = "$" + String(amount) **显式地**将值 199.98 转换为字符串，并在开头加上字符 "$"。此时，amount 持有字符串值 "$199.98"，所以第二个 console.log(..) 语句打印输出时就不需要转换类型了。

JavaScript 开发者应该注意到变量 amount 表示值 99.99、199.98 和 "$199.98" 的这种灵活性。静态类型的狂热支持者可能会单独使用一个变量，例如，使用 amountStr 来保存最后的 "$199.98"，因为这是一个不同的类型。

无论是哪一种方式，你都会注意到 amount 保存的值会随着程序运行而有所变化，这展示了变量的主要用途：管理程序**状态**。

换句话说，**状态**跟踪了值随着程序运行的变化。

变量的另一个常见用法是集中设置值。更常见的说法是**常量**，即声明一个变量，赋予一个特定值，然后这个值在程序执行过程中保持不变。

这些常量的声明通常放在程序的开头，所以如果需要改变这些值的话，那么就会有一个很方便的集中位置。通常来说，在 JavaScript 中作为常量的变量用大写表示，多个单词之间用下划线 _ 分隔。

以下是一个简单的示例：

```
var TAX_RATE = 0.08; // 8%的营业税

var amount = 99.99;

amount = amount * 2;

amount = amount + (amount * TAX_RATE);
```

```
console.log( amount );              // 215.9784
console.log( amount.toFixed( 2 ) ); // "215.98"
```

> console.log(..) 是作为 console 值的一个对象属性的函数 log(..)，与此类似，toFixed(..) 是一个可以通过数字值访问的函数。JavaScript 的数字不会自动格式化为美元表示法，因为引擎无法了解你的意图，也没有适合现金的类型。toFixed(..) 可以帮助我们指定保留数字小数点后的几位，并按照期望生成字符串值。

变量 TAX_RATE 是依靠惯例而定的一个**常量**，程序中没有任何特殊实现可以防止它被修改。而如果这个城市的营业税提高到 9%，那么我们可以很容易地修改程序，只需要在唯一一处修改 TAX_RATE 值为 0.09，而不是在程序中搜索多个 0.08，然后修改所有的值。

在编写本部分时，最新版本的 JavaScript（一般被称为“ES6”）提供了一个新的常量声明方法，使用 const 代替了 var：

```
// 自ES6起：
const TAX_RATE = 0.08;

var amount = 99.99;

// ..
```

常量和值不变的变量一样有用，而且常量还可以防止值在最初设定后被无意修改。如果想要在初始声明后给 TAX_RATE 赋其他值，那么程序会拒绝这个修改（严格模式下会失败退出，参见 2.4 节）。

另外，这种防止出错的“保护”措施与静态类型相似，所以你应该可以理解其他语言中的静态类型是多么具有吸引力了！

> 有关如何在程序中使用变量中的不同值，参见本系列《你不知道的 JavaScript（中卷）》第一部分中的前两章。

1.8 块

当你选购好新手机而结账时，手机商店的店员必须完成一系列的步骤。

与此类似，我们常常需要将在代码中的一系列语句组织到一起，这些语句通常被称为**块**。在 JavaScript 中，使用一对大括号 { .. } 在一个或多个语句外来表示块。考虑：

```
var amount = 99.99;
```

```
//一个通用的块
{
    amount = amount * 2;
    console.log( amount );   // 199.98
}
```

这种独立的 { .. } 块是合法的，但在 JavaScript 程序中比较少见。通常来说，块会与其他某个控制语句组合在一起，比如 if 语句（参见 1.9 节）或循环（参见 1.10 节）。举例来说：

```
var amount = 99.99;

// amount是否足够大呢?
if (amount > 10) {            // <-- 块与if组合
    amount = amount * 2;
    console.log( amount );   // 199.98
}
```

我们将在下一节中介绍 if 语句，但正如你可以看到的，包含两个语句的 { .. } 块与 if (amount > 10) 结合在一起了；块内的语句只有在条件判断成立时才会运行。

> 与 console.log(amount); 这样的大多数其他语句不同，块语句不需要以分号（;）结尾。

1.9　条件判断

“您想要再加一个价值 $9.99 的屏幕保护膜吗？”手机商店的店员这么问就是在请你作出一个决定。你可能会先看看钱包或银行账号的当前状态再回答这个问题。但显然，这只是一个简单的“是”或“否”的问题。

程序中有很多种方法可以用于表示**条件判断**（也就是**决策**）。

最常用的是 if 语句。本质上就是在表达“如果这个条件是真的，那么进行后续这些……”。举例来说：

```
var bank_balance = 302.13;
var amount = 99.99;

if (amount < bank_balance) {
    console.log( "I want to buy this phone!" );
}
```

if 语句要求在括号 ( ) 中放一个表达式，这个表达式要么是 true，要么是 false。在这个程序中，我们提供的表达式是 amount < bank_balance，根据 bank_balance 变量中的数量，其求

值结果确实是 `true` 或者 `false`。

你还可以提供一个用于 `if` 条件不为真时的选择，我们将其称为 `else` 语句。考虑：

```
const ACCESSORY_PRICE = 9.99;

var bank_balance = 302.13;
var amount = 99.99;

amount = amount * 2;

// 是否可以提供额外的购买？
if ( amount < bank_balance ) {
     console.log( "I'll take the accessory!" );
     amount = amount + ACCESSORY_PRICE;
}
// 否则：
else {
    console.log( "No, thanks." );
}
```

在以上的示例中，如果 `amount < bank_balance` 为真，那么就会打印出 `"I'll take the accessory!"`，并在变量 `amount` 上加上 `9.99`。否则，`else` 语句就会礼貌地回答 `"No, thanks."`，并保持 `amount` 不变。

正如我们在 1.5 节中讨论的那样，不满足期望类型的值通常会被强制转换为需要的类型。`if` 语句需要布尔型的值，如果传递的值是非布尔型的，那么就会发生类型转换。

JavaScript 定义了一系列特定的值，这些值在强制转换为布尔型时会被认为是“假的”，它们会转化为 `false`，其中包括 `0` 和 `""` 这样的值。任何不在这个列表中的其他值会自动成为“真的”，因此在强制转换为布尔型时会转化为 `true`。真值包括 `99.99` 和 `"free"` 这样的值。要想获得更多信息，参见 2.1.3 节中的“真与假”部分。

除 `if` 之外，还有其他形式的**条件判断**。比如，`switch` 语句可以用作一组 `if..else` 语句的简写（参见第 2 章）。循环（参见 1.10 节）通过一个**条件判断**来确定是继续还是停止。

> 有关测试条件判断时可能隐式发生的类型转换的更多信息，参见本系列《你不知道的 JavaScript（中卷）》第一部分中的第 4 章。

1.10　循环

商店忙碌时会有一个等候服务的顾客队列。如果这个队列中还有顾客，那么店员就要继续为下一位顾客服务。

重复一系列动作，直到不满足某个条件，换句话说，重复只发生在满足条件的情况下，这

就是程序循环的工作；循环有多种形式，但都满足基本的行为特性。

循环包括测试条件以及一个块（通常就是 `{ .. }`）。循环块的每次执行被称为一个**迭代**。

比如，`while` 循环和 `do..while` 循环形式展示了重复一个语句块直到一个条件判断求值不再为真这个概念：

```
while (numOfCustomers > 0) {
    console.log( "How may I help you?" );

    // 帮助顾客……

    numOfCustomers = numOfCustomers - 1;
}

// 对比：

do {
    console.log( "How may I help you?" );

    // 帮助顾客……

    numOfCustomers = numOfCustomers - 1;
} while (numOfCustomers > 0);
```

这些循环之间的唯一实际区别是，条件判断在第一次迭代执行前（`while`）检查还是在第一次迭代后（`do..while`）检查。

不管是哪种形式，如果条件判断测试结果为 `false`，那么都不会运行下一轮迭代。这意味着，如果第一次的条件判断为 `false`，那么 `while` 循环就不会执行，而 `do..while` 循环只会运行一次。

有时循环要在一组数字上迭代，比如从 `0` 到 `9`（10 个数字）。你可以设置一个 `i` 这样的循环迭代变量，初始为 `0`，然后每次迭代增加 `1`。

> 出于多种历史原因，编程语言几乎总是从零开始计数，也就是说，是从 `0` 而不是从 `1` 开始。如果不熟悉这种思维模式的话，一开始可能会感到非常迷惑。花点时间进行从 `0` 开始的计数训练，并适应这一点。

条件判断会在每次迭代时测试，这就好像是在循环内部有一个隐式的 `if` 语句。

我们可以通过 JavaScript 的 `break` 语句来结束循环。另外，我们也可以看到，很容易就会创建出一个如果不使用 `break` 机制就会陷入死循环的循环。

举例来说：

```
var i = 0;
```

```
// while..true循环会永久运行，不是吗?
while (true) {
    // 停止循环?
    if ((i <= 9) === false) {
        break;
    }

    console.log(i);
    i = i + 1;
}
// 0 1 2 3 4 5 6 7 8 9
```

> 这并非是你在自己的程序中一定要效仿的一个实际循环形式。在这里展示只是为了说明问题。

虽然使用 `while`（或者 `do..while`）循环也可以手动完成任务，但还有一个专门为此设计的语法形式，我们将其称为 `for` 循环：

```
for (var i = 0; i <= 9; i = i + 1) {
    console.log( i );
}
// 0 1 2 3 4 5 6 7 8 9
```

在上述的示例中可以看到，两种情况下条件 `i <= 9` 对于前十次迭代都为 `true`，但当 `i` 为 `10` 时会变为 `false`。

`for` 循环有 3 个分句：初始化分句（`var 1 = 0`）、条件测试分句（`i <= 9`），以及更新分句（`i = i + 1`）。所以，如果你需要在循环迭代中计数，那么最紧凑、最容易理解和编写的形式就是 `for` 循环。

还有其他一些特殊的循环形式，专门用于在特定的值上迭代，比如对象属性（参见第 2 章），其中隐式的测试条件为是否所有的属性都已经处理完毕。无论循环的形式是什么，“循环到条件为否”这个概念是保持不变的。

1.11 函数

手机商店的店员应该不会随身携带计算器来计算税费和最后的应付金额。这个任务是她需要定义一次并多次复用的。很有可能的是，公司的收银设备（计算机或平板等）内置了这样的“函数”。

类似地，你的程序也几乎总是需要将代码的任务分割成可复用的片段，而不是一直重复编码。实现这一点的方法就是定义一个**函数**。

通常来说，函数是可以通过名字被“调用”的已命名代码段，每次调用，其中的代码就会运行。考虑：

```
function printAmount() {
    console.log( amount.toFixed( 2 ) );
}

var amount = 99.99;

printAmount(); // "99.99"

amount = amount * 2;

printAmount(); // "199.98"
```

函数可以接受参数，即你传入的值，也可以返回一个值：

```
function printAmount(amt) {
    console.log( amt.toFixed( 2 ) );
}

function formatAmount() {
    return "$" + amount.toFixed( 2 );
}

var amount = 99.99;

printAmount( amount * 2 );    // "199.98"

amount = formatAmount();
console.log( amount );    // "$99.99"
```

函数 `printAmount(..)` 接受一个名为 `amt` 的参数。函数 `formatAmount()` 返回一个值。你也可以在同一个函数中同时使用这两种技术。

通常来说，你会在计划多次调用的代码上使用函数，但只是将相关的代码组织到一起成为命名集合也是很有用的，即使只准备调用一次。

考虑：

```
const TAX_RATE = 0.08;

function calculateFinalPurchaseAmount(amt) {
    // 根据税费来计算新的数值
    amt = amt + (amt * TAX_RATE);

    // 返回新的数值
    return amt;
}

var amount = 99.99;
```

```
amount = calculateFinalPurchaseAmount( amount );

console.log( amount.toFixed( 2 ) );      // "107.99"
```

尽管 calculateFinalPurchaseAmount(..) 只被调用了一次，但将它的行为组织到一个独立的命名函数中使得使用其逻辑（amount = calculateFinal... 语句）的代码更为清晰。函数中的语句越多，其效果就会越明显。

作用域

如果你向手机商店的店员询问的手机型号是该商店里没有的，那么她也就无法将你想要的手机卖给你。她只能接触到商店里现有的手机。你也就不得不到另外一家商店尝试看看能否找到你想要的手机型号了。

编程中的一个术语可以表示这个概念：**作用域**（严格说是**词法作用域**）。在 JavaScript 中，每个函数都有自己的作用域。作用域基本上是变量的一个集合以及如何通过名称访问这些变量的规则。只有函数内部的代码才能访问这个函数作用域中的变量。

同一个作用域内的变量名是唯一的，所以不能有两个变量 a 一个接一个地放在一起。但是，同一个变量名 a 可以出现在不同的作用域中：

```
function one() {
    // 这个a只属于one()函数
    var a = 1;
    console.log( a );
}

function two() {
    // 这个a只属于two()函数
    var a=2;
    console.log( a );
}

one();      // 1
two();     // 2
```

此外，作用域是可以彼此嵌套的，就好像生日聚会上的小丑可以在一个气球内部吹起另一个气球那样。如果一个作用域嵌套在另外一个作用域内，那么内层作用域中的代码可以访问这两个作用域中的变量。

考虑：

```
function outer() {
    var a = 1;

    function inner() {
        var b = 2;
```

```
        // 这里我们既可以访问a，也可以访问b
        console.log( a + b );    // 3
    }

    inner();

    // 这里我们只能访问a
    console.log( a );         // 1
}

outer();
```

词法作用域的规则表明，一个作用域内的代码可以访问这个作用域内以及任何包围在它之外的作用域中的变量。

因此，`inner()` 函数内部的代码可以访问变量 `a` 和 `b`，但是 `outer()` 中的代码只能访问 `a`，不能访问 `b`，因为这个变量 `b` 只在 `inner()` 函数内部。

我们来回顾一下前面的代码片段：

```
const TAX_RATE = 0.08;

function calculateFinalPurchaseAmount(amt) {
    // 根据税费来计算新的数值
    amt = amt + (amt * TAX_RATE);

    // 返回新的数值
    return amt;
}
```

因为词法作用域的缘故，我们可以从函数 `calculateFinalPurchaseAmount(..)` 的内部访问常量（变量）`TAX_RATE`，尽管我们并没有将它传递进去。

> 有关词法作用域的更多信息，参见本系列《你不知道的 JavaScript（上卷）》第一部分中的前三章。

1.12　实践

在编程学习中，实践是绝对无法替代的。无论我如何在这里阐释说明，理论都无法让你成为一个程序员。

谨记这一点。我们来尝试针对在本章中学习到的概念进行一些练习。我会给出“需求”，你先试着解决这些“需求”。然后查看以下列出的代码，看看我是如何解决的。

- 编写程序以计算购买手机所需的总金额。你需要一直购买手机（提示：循环！）直到银行账号中的资金不足。而且，只要价格低于你的心理预期值，那么就要为每个手机购买附件。

- 计算总金额后再加上税费，然后以适当的格式打印出计算出的总金额。
- 最后，检查银行账号的余额，确认是否能买得起。
- 需要为“税率”、“手机价格”、“附件价格”和“预算阈值”建立一些常量，为“银行账号的余额”建立变量。
- 你应该定义一些函数来计算税费，格式化价格加上“$”符号并保留两位小数。
- **附加题**：试着在这个程序中集成输入，你可以使用 1.3.2 节中介绍的 `prompt(..)`。比如，你可以提示用户输入他们的银行账号余额。享受吧，发挥你的创造力！

好了，你现在可以开始实践了。在你自己尝试之前不要先看我的代码！

> 因为本书是一本关于 JavaScript 的书，显然我会使用 JavaScript 来完成这个练习。但你也可以根据个人意愿而使用其他语言来实现。

以下是我针对上述练习而设计的 JavaScript 解决方案：

```
const SPENDING_THRESHOLD = 200;
const TAX_RATE = 0.08;
const PHONE_PRICE = 99.99;
const ACCESSORY_PRICE = 9.99;

var bank_balance = 303.91;
var amount = 0;

function calculateTax(amount) {
    return amount * TAX_RATE;
}

function formatAmount(amount) {
    return "$" + amount.toFixed( 2 );
}

// 如果还有余额，那么继续购买手机
while (amount < bank_balance) {
    // 购买新的手机！
    amount = amount + PHONE_PRICE;

    // 是否可以负担得起附件？
    if (amount < SPENDING_THRESHOLD) {
        amount = amount + ACCESSORY_PRICE;
    }
}

// 别忘了交税
amount = amount + calculateTax( amount );

console.log(
    "Your purchase: " + formatAmount( amount )
```

```
);
// 你的购买金额：$334.76

// 你真的可以负担得起本次购买吗？
if (amount > bank_balance) {
    console.log(
        "You can't afford this purchase. :("
    );
}
// 你无法负担本次购买。 :(
```

> 运行这个 JavaScript 程序最简单的方法是，将其输入到你手边浏览器的开发者终端中。

你是如何实现的呢？你现在已经看到了我的代码，不妨再试一下。你可以修改其中的一些常量，看看这个程序在不同的值之下是怎么运行的。

1.13　小结

学习编程并不必然是复杂、费力的过程。你需要熟悉几个基本的概念。

这些概念如同积木。要想构造高塔，首先要将积木一层层摞在一起。编程也是如此。以下是一些核心的编程积木块。

- 在值上执行动作需要**运算符**。
- 执行各种类型的动作需要**值和类型**，比如，对数字进行数学运算，用字符串输出。
- 在程序的执行过程中需要**变量**来保存数据（也就是**状态**）。
- 需要 `if` 这样的条件判断来作出决策。
- 需要循环来重复任务，直到不满足某个条件。
- 需要**函数**将代码组织为逻辑上可复用的块。

代码注释是编写可读代码的一种有效方法，能让你的代码更易于理解和维护，如果以后出现问题的话也更加容易进行修复。

最后，不要忽略练习的威力，学习如何编写代码的最好方法就是不断编写代码。

很高兴你已经在开始学习如何编码的路上了！继续前进吧！不要忘了查阅其他的编程入门资源（书籍、博客和在线培训等）。本章和本部分就是一个很好的起点，但它们仅仅只是一个概要介绍。

下一章将会介绍本章出现的多个概念，但会从 JavaScript 的角度来解释，这会强调多个主要的主题，这些主题是在本系列其他书中深入详细介绍的。

第 2 章

深入 JavaScript

我们在前一章中介绍了编程的基本组件，如变量、循环、条件判断和函数。当然，其中展示的所有代码都是用 JavaScript 语言编写的。而本章的主要关注点是作为 JavaScript 开发者在开始编写 JavaScript 代码时所需要了解的知识。

我们将在本章中介绍很多概念，这些概念需要阅读其他的“你不知道的 JavaScript”系列图书才能完全掌握。你可以将本章看作本系列其他图书将要详细介绍的主题的概论。

尤其重要的一点是，如果你还只是 JavaScript 方面的新手，那么应该多花点时间查看这些概念并反复练习本章中的示例代码。坚固的基础都是一点一点构造起来的，因此不要期望第一次阅读就能够马上完全理解这些概念。

深入学习 JavaScript 的旅程这就开始了。

> 正如我在第 1 章中所说的，在阅读和学习本章的过程中，你绝对应该亲自试验一下所有的代码示例。记住，本章中的部分代码假定了编写本部分时 JavaScript 最新版本（JavaScript 规范的官方名称 ECMAScript 第 6 版，一般称为“ES6”）中引入的功能。如果你恰好在使用 ES6 前的旧版浏览器，那么这些代码可能无法正常运行。你应该使用浏览器（如 Chrome、Firefox 或 IE 等）的更新版本。

2.1　值与类型

我们在第 1 章中已经提到过，JavaScript 的值有类型，但变量无类型。以下是可用的内置类型：

- 字符串
- 数字
- 布尔型
- null 和 undefined
- 对象
- 符号（ES6 中新增的）

JavaScript 提供了一个 typeof 运算符，该运算符可以用来查看值的类型：

```
var a;
typeof a;           // "undefined"

a = "hello world";
typeof a;           // "string"

a=42;
typeof a;           // "number"

a = true;
typeof a;           // "boolean"

a = null;
typeof a;           // "object"--诡异，这是bug

a = undefined;
typeof a;           // "undefined"


a={b:"c"};
typeof a;           // "object"
```

typeof 运算符的返回值永远是这 6 个（对 ES6 来说是 7 个）字符串值之一。也就是说，typeof "abc" 返回 "string"，而不是 string。

请注意这段代码中的变量是如何持有多个不同类型的值的，和表面看起来不同，typeof 并不是在询问“a 的类型”，而是“a 中当前值的类型”。在 JavaScript 中，只有值有类型；变量只是这些值的容器。

typeof null 是一个有趣的示例，你期望它返回的会是 "null"，但它返回的却是 "object"，这大概会让你觉得很意外。

这是 JavaScript 中存在已久的一个 bug，也似乎是一个永远不会被修复的 bug。Web 上的太多代码都依赖于这个 bug，因此，修复它会导致大量的新 bug！

另外，注意 a = undefined。我们显式地将 a 的值设置为 undefined，但从行为上来说，这

与这段代码最开始的 var a 这样还未赋值的变量是一样的。变量能够以几种不同的方式进入这样"未定义"值的状态，其中包括没有返回值的函数和使用 void 运算符。

2.1.1 对象

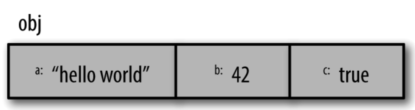
对象类型是指一个组合值，你可以为其设置属性（命名的位置），每个属性可以持有属于自己的任意类型的值。这也许是 JavaScript 中所有的值类型中最有用的一个：

```
var obj = {
    a: "hello world",
    b: 42,
    c: true
};

obj.a;      // "hello world"
obj.b;      // 42
obj.c;      // true

obj["a"];   // "hello world"
obj["b"];   // 42
obj["c"];   // true
```

可以将这个 obj 值想象成以下这个可视化的状态，这样更便于理解：



可以通过**点号**（如 obj.a 所示）或者**中括号**（如 obj["a"] 所示）来访问属性。点号更简短易读，因而尽量使用这种方式。

如果属性名中有特殊字符的话，那么中括号表示法就会很有用，如 obj["hello world!"]，在通过中括号表示法访问时，我们通常将这样的属性称为**键值**。[ ] 表示法接受变量（后面将会解释）或者字符串字面值（需要使用 ".." 或 '..' 包裹）。

当然，如果需要访问的某个属性 / 键值的名称保存在另一个变量中时，括号表示法也很有用，如下所示：

```
var obj = {
    a: "hello world",
    b: 42
};

var b = "a";

obj[b];       // "hello world"
obj["b"];     // 42
```

有关 JavaScript 对象的更多信息，参见本系列《你不知道的 JavaScript（上卷)》第二部分，特别是第 3 章。

在 JavaScript 程序中，你还需要经常和其他两个值类型打交道：**数组**和**函数**。但你更应该将它们看作是对象类型的特殊子类型，而不是内置类型。

1. 数组

数组是一个持有（任意类型）值的对象，这些值不是通过命名属性 / 键值索引，而是通过数字索引位置。如下所示：

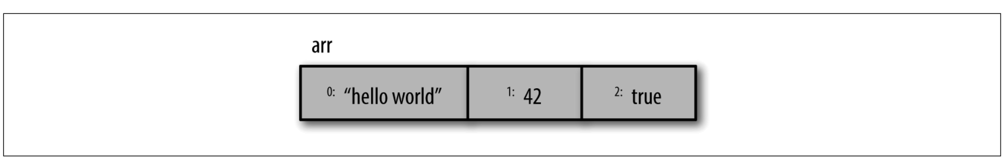
```
var arr = [
    "hello world",
    42,
    true
];

arr[0];         // "hello world"
arr[1];         // 42
arr[2];         // true
arr.length;     // 3

typeof arr;     // "object"
```

像 JavaScript 这样从零开始计数的语言，会使用 `0` 作为数组中第一个元素的索引。

可以将这个 `arr` 值想象成以下这种可视化的状态，这样更便于理解：



因为数组是特殊的对象（正如 `typeof` 所暗示的那样），所以它们也可以有属性，其中包括自动更新的 `length` 属性。

从理论上来说，你可以将数组当作普通的对象来使用，为其添加自己的命名属性，或者你也可以只为一个对象提供数字属性（`0`、`1` 等），就像数组一样使用它。但一般来说，这样使用这些类型是不合理的。

最好的同时也是最自然的方法就是使用数字位置索引数组，通过命名属性使用对象。

2. 函数

在 JavaScript 程序中，另一个常用的对象子类型是函数：

```
function foo() {
    return 42;
}

foo.bar = "hello world";

typeof foo;         // "function"
typeof foo();       // "number"
typeof foo.bar;     // "string"
```

函数也同样是对象的一个子类型，因为 `typeof` 返回 `"function"`，这意味着 `function` 是一个主类型，因此，`function` 可以拥有属性，但通常只在很少的情况下才会使用函数的对象属性（如 `foo.bar`）。

> 有关 JavaScript 值和类型的更多信息，参见本系列《你不知道的 JavaScript（中卷）》第一部分的前两章。

2.1.2　内置类型方法

我们刚刚讨论过的内置类型和子类型拥有作为属性和方法暴露出来的行为，这是非常强大有力的功能。

举例来说：

```
var a = "hello world";
var b = 3.14159;

a.length;               // 11
a.toUpperCase();        // "HELLO WORLD"
b.toFixed(4);           // "3.1416"
```

能够调用 `a.toUpperCase()` 的原理要比值上存在的方法这种解释复杂得多。

简单地说，存在一个 `String`（`S` 大写）对象封装形式，通常称为“原生的”，与基本 `string` 类型相对应；这个对象封装在其原型中定义了 `toUpperCase()` 方法。

像前面的代码片段中那样将 `"hello world"` 这样的原生值作为对象使用时，在引用其属性和方法时（比如 `toUpperCase()`），JavaScript 会（暗自）自动地将这个值“封箱”为其对应的对象封装。

字符串值可以封装为 `String` 对象，数字可以封装为 `Number` 对象，布尔型值可以封装为 `Boolean` 对象。在多数情况下，你不需要思考直接使用这样的值的对象封装形式，所有情

况下都使用原生值形式，让 JavaScript 来负责其余的事情吧。

> 有关 JavaScript 原生类型和“封箱”的更多信息，参见本系列《你不知道的 JavaScript（中卷）》第一部分的第 3 章。要想更好地理解一个对象的原型，参见《你不知道的 JavaScript（上卷）》第二部分的第 5 章。

2.1.3 值的比较

JavaScript 程序中有两种主要的值比较：**相等**与**不等**。不管比较的类型是什么，任何比较的结果都是严格的布尔值（`true` 或者 `false`）。

1. 类型转换

我们在第 1 章中简单地讨论了类型转换，现在再深入讨论一下。

JavaScript 中有两种类型转换：**显式的**类型转换与**隐式的**类型转换。显式的类型转换就是你可以在代码中看到的类型由一种转换到另一种，而隐式的类型转换多是某些其他运算可能存在的隐式副作用而引发的类型转换。

你可能听过“类型转换是邪恶的”这种说法，这显然是因为有些情况下的类型转换确实会产生一些出人意料的结果。最能够激怒程序员的事情就是语言发生出乎意料的变化。

类型转换并不是邪恶的，也并不一定是出人意料的。实际上，使用类型转换的多数情况都是非常容易理解的，甚至可以提高代码的可读性。我们不再深入这个有争议的话题，本系列《你不知道的 JavaScript（中卷）》第一部分的第 4 章覆盖了这个主题的方方面面。

以下是显式类型转换的一个示例：

```
var a = "42";

var b = Number( a );

a;              // "42"
b;              // 42--数字!
```

以下是隐式类型转换的一个示例：

```
var a = "42";

var b = a * 1;  // 这里"42"隐式地转换为了42

a;              // "42"
b;              // 42--数字!
```

2. 真与假

我们在第 1 章中简单地提到了值的“真”与“假”的特性：当非布尔型的值被强制转换为

布尔型时，结果是 true 还是 false 呢？

JavaScript 中“假”值的详细列表如下：

- ""（空字符串）
- 0、-0、NaN(无效数字)
- null、undefined
- false

任何不在“假”值列表中的值都是“真”值。以下是一些示例：

- "hello"
- 42
- true
- [ ]、[1, "2", 3]（数组）
- { }、{ a: 42 }（对象）
- function foo() { .. }（函数）

你需要记住非常重要的一点，只有在非布尔型值强制转换为布尔型值时才会遵从这个“真”/“假”转换规则。看起来是将一个值转换成布尔型，而实际上并没有，这样的情况还是很容易令人迷惑的。

3. 相等

相等运算符有四种：==、===、!= 和 !==。! 形式显然是相应的“不等”版本；不要混淆了**不等关系**和**不相等**。

== 和 === 的区别在于，== 检查值相等，而 === 检查值和类型相等。但这么说并不精确。正确的说法是，== 检查的是允许类型转换情况下的值的相等性，而 === 检查不允许类型转换情况下的值的相等性；因此，=== 经常被称为“严格相等”。

思考以下的情况，== 的粗略相等比较允许隐式的类型转换，而严格相等比较 === 则不允许：

```
var a = "42";
var b = 42;

a == b;         // true
a === b;        // false
```

在比较 a == b 的过程中，JavaScript 注意到这两个值的类型不匹配，于是它会按照一系列的顺序步骤将其中一个或两个值的类型转换到其他类型，直到类型匹配，然后进行简单的值相等检查。

思考一下这个过程，有两种类型转换情况可以导致 `a == b` 结果为真。最终比较的要么是 `42 == 42`，要么是 `"42" == "42"`，那么到底是哪一个呢？

答案是，"42" 被转化为了 42，使得最终的比较为 `42 == 42`。在这个简单的示例中，过程似乎是无关紧要的，因为最终结果都是一样的。而在一些更为复杂的示例中，不只是最终的比较结果很重要，转换过程也会产生影响。

`a === b` 为假，因为类型转换不被允许，所以简单的值比较的结果显然为假。很多开发者认为 === 更可预测，所以他们支持一直使用 === 而避免使用 ==。我认为这种观点是很短视的。在我看来，== 是一个强大的工具，**如果花时间来学习其工作原理的话**，那么对程序是很有益的。

我们不打算面面俱到地覆盖 == 比较中类型转换的所有工作细节。多数情况都是比较容易理解的，但也有一些重要的特例需要小心对待。你可以查看 ES5 规范（http://www.ecma-international.org/ecma-262/5.1/）的 11.9.3 节来了解精确的规则，与围绕着 == 的负面传闻相比，你可能会吃惊于这套机制看起来是多么直观。

下面我将列出几条简单的规则，将所有这些大量的细节归结为简单的条目，以帮助你了解在不同情况下应该使用 == 还是 ===。

- 如果要比较的两个值的任意一个（即一边）可能是 `true` 或者 `false` 值，那么要避免使用 ==，而使用 ===。
- 如果要比较的两个值中的任意一个可能是特定值（`0`、`""` 或者 `[]`——空数组），那么避免使用 ==，而使用 ===。
- 在所有其他情况下，使用 == 都是安全的。不仅仅只是安全而已，这在很多情况下也会简化代码，提高代码的可读性。

提炼出的这几条规则要求你认真思考自己的代码，思考要比较相等性的变量的可能值有哪些。如果你能够确定这些值，并且 == 是安全的，那么就可以使用它！如果不能确定其值，那么就使用 ===。就是这么简单。

不等 `!=` 与 == 对应，`!==` 与 === 对应。前面讨论过的所有规则和观察对这些不等比较也都是适用的。

如果是比较两个非原生值的话，比如对象（包括函数和数组），那么你需要特殊注意 == 与 === 这些比较规则。因为这些值通常是通过引用访问的，所以 == 和 === 比较只是简单地检查这些引用是否匹配，而完全不关心其引用的值是什么。

举例来说，通过简单地在元素之间插入逗号（`,`），数组在默认情况下会转换为字符串。你可能会认为内容相同的两个数组也会 == 相等，但并非如此：

```
var a = [1,2,3];
var b = [1,2,3];
var c = "1,2,3";

a == c;      // true
b == c;      // true
a == b;      // false
```

> 有关 == 相等比较规则的更多信息，参见 ES5 规范（11.9.3 节）以及本系列《你不知道的 JavaScript（中卷）》第一部分的第 4 章；有关值与引用的更多信息，参见《你不知道的 JavaScript（中卷）》第一部分的第 2 章。

4. 不等关系

运算符 <,>、<= 和 >= 用于表示不等关系，在规范中被称为“关系比较”。通常来说，它们用于比较有序值，比如数字。`3 < 4` 这样的比较是很容易理解的。

也可以比较 JavaScript 中的字符串值的不等关系，这是按照常见的字母表规则来比较的（`"bar" < "foo"`）。

类型转换呢？与 == 比较规则类似（尽管是不完全相同！）的规则可以应用于不等关系运算符。注意，并没有与“严格相等”=== 类似的、不允许类型转换的“严格不等关系”运算符。

考虑：

```
var a = 41;
var b = "42";
var c = "43";

a < b;       // true
b < c;       // true
```

上述代码发生了什么？ ES5 规范中的 11.8.5 节中提到，如果 < 比较的两个值都是字符串，就像在 `b < c` 中那样，那么比较按照字典顺序（即字典中的字母表顺序）进行。如果其中一边或两边都不是字符串，就像在 `a < b` 中那样，那么这两个值的类型都转换为数字，然后进行普通的数字比较。

针对类型可能不同的值之间的比较，请记住，没有“严格不等”形式可用。你最需要了解的一点是，当其中一个值无法转换为有效数字时的情形，如下所示：

```
var a = 42;
var b = "foo";

a < b;      // false
a > b;      // false
a == b;     // false
```

为什么这三个比较结果都为假呢？这是因为 < 和 > 比较中的值 `b` 都被类型转换为了“无效数字值” `NaN`，规范设定 `NaN` 既不大于也不小于任何其他值。

`==` 比较的结果为假的原因则不同。不论解释为 `42 == NaN` 还是 `"42" == "foo"`，都会使得 `a == b` 结果为假——我们已经在前文中介绍过，这种情况属于前者。

> 有关不等比较规则的更多信息，参见 ES5 标准的 11.8.5 节以及本系列《你不知道的 JavaScript（中卷）》第一部分的第 4 章。

2.2 变量

在 JavaScript 中，变量的名称（包括函数名称）必须是有效的**标识符**。考虑到 Unicode 这样的非传统字符的情况，标识符中有效字符的严格完整规则有点复杂。如果只是考虑常用的 ASCII 字母数字的话，那么规则是非常简单的。

标识符必须由 `a~z`、`A~Z`、`$` 或 `_` 开始。它可以包含前面所有这些字符以及数字 `0~9`。

一般来说，变量标识符的这个规则对属性命名也同样适用。但是，有些单词不能用作变量名，但可以作为属性名。这些单词被称为“保留词”，其中包括 JavaScript 关键字（`for`、`in`、`if` 等）以及 `null`、`true` 和 `false`。

> 有关保留字的更多信息，参见本系列《你不知道的 JavaScript（中卷）》第一部分的附录 A。

函数作用域

如果使用关键字 `var` 声明一个变量，那么这个变量就属于当前的函数作用域，如果声明是发生在任何函数外的顶层声明，那么这个变量则属于全局作用域。

1. 提升

无论 `var` 出现在一个作用域中的哪个位置，这个声明都属于整个作用域，在其中到处都是可以访问的。

这一行为被比喻地称为**提升**（hoisting），`var` 声明概念上“移动”到了其所在作用域的最前面。从技术上来说，可以通过如何编译代码更精确地解释这个过程，我们暂时先不赘述这些细节。

考虑：

```
var a = 2;
```

```
foo();                  // 因为foo()而运行
                        // 声明是“被提升的”

function foo() {
    a = 3;

    console.log( a );   // 3

    var a;              // 声明是“被提升的”
                        // 到foo()的顶端
}

console.log( a );       // 2
```

> 在变量声明出现之前，依靠变量**提升**在其作用域使用这个变量并不常见，也并不是一个好的想法；这样的代码可能会令人非常迷惑。相比之下，使用**提升后的**函数声明则要常见得多，就像我们在 `foo()` 的正式声明出现前就调用了它。

2. 嵌套作用域

声明后的变量在这个作用域内是随处可以访问的，包括所有低层 / 内层的作用域。举例来说：

```
function foo() {
    var a = 1;

    function bar() {
        var b = 2;

        function baz() {
            var c = 3;

            console.log( a, b, c );  // 1 2 3
        }

        baz();
        console.log( a, b );         // 1 2
    }

    bar();
    console.log( a );                // 1
}

foo();
```

注意，在上述示例中，`c` 在 `bar()` 的内部是不可访问的，因为它只声明在内层 `baz()` 作用域，`b` 在 `foo()` 中是不可访问的，也是同样的原因。

如果试图在一个作用域中访问一个不可访问的变量，那么就会抛出 `ReferenceError`。如果

试图设定尚未声明的变量，那么就会导致在顶层全局作用域创建这个变量（不好！）或者出现错误，这要根据是否处于“严格模式”而定（参见 2.4 节）。我们来看一下：

```
function foo() {
    a = 1;  // a没有正式声明
}

foo();
a;          // 1--哎呀，自动全局变量 :(
```

这是一个很差的实践。不要这么做！一定要正式声明你的变量。

除了在函数层级声明变量，ES6 还支持通过 `let` 关键字声明属于单独块（`{ .. }` 对）的变量。除了细节上有一些微小差别，这里的作用域规则和我们在函数中看到的基本上是相同的：

```
function foo() {
    var a = 1;

    if (a >= 1) {
        let b = 2;

        while (b < 5) {
            let c = b * 2;
            b++;

            console.log( a + c );
        }
    }
}

foo();
// 5 7 9
```

因为使用了 `let` 而不是 `var`，所以 `b` 只属于 `if` 语句，不属于整个 `foo()` 函数的作用域。与此类似，`c` 只属于 `while` 循环。块作用域非常有助于更细化地管理变量作用域，从而更容易随着时间的发展而维护代码。

> 有关作用域的更多信息，参见本系列中的《你不知道的 JavaScript（上卷）》第一部分。有关 `let` 块作用域的更多信息，参见本书第二部分。

2.3 条件判断

除了第 1 章中简单介绍过的 `if` 语句，JavaScript 还提供了几种其他条件判断机制，我们也应该了解一下。

你有时可能会编写出一系列的 if..else..if 语句，如下所示：

```
if (a == 2) {
    // 做某件事情
}
else if (a == 10) {
    // 做另一件事情
}
else if (a == 42) {
    // 做另一件事情
}
else {
    // 反馈到这儿
}
```

这样的结构可以运行，但是有点繁复，因为你需要为每种情况都指定一个测试。switch 语句是另一种选择：

```
switch (a) {
    case 2:
        // 做某件事情
        break;
    case 10:
        // 做另一件事情
        break;
    case 42:
        // 做另一件事情
        break;
    default:
        // 反馈到这儿
}
```

如果只想要运行某个 case 下的语句，那么 break 是很重要的。如果某个 case 省略了 break，而这个 case 匹配或运行的话，那么会一直执行到下一个 case 的语句，不管那个 case 是否匹配。这种所谓的“通过（fall through）”有时是很有用的。

```
switch (a) {
    case 2:
    case 10:
        // 某个很棒的东西
        break;
    case 42:
        // 其他东西
        break;
    default:
        // 反馈
}
```

在上述示例中，如果 2 或者 10 匹配的话，那么就会执行“某个很棒的东西”代码语句。

在 JavaScript 中，条件判断的另一种形式是“条件运算符”，通常被称为“三进制运算符”。它更像是单个 if..else 语句的紧凑版，如下所示：

```
var a = 42;

var b = (a > 41) ? "hello" : "world";

// 与以下类似:

// if (a > 41) {
//    b = "hello";
// }
// else {
//    b = "world";
// }
```

如果条件表达式（这里是 a > 41）求值为真，那么就会返回第一个子句（"hello"）；否则，结果就是第二个子句（"world"），不论结果是什么，都会赋给 b。

条件运算符并不一定要用在赋值上，但这肯定是最常见的用法。

> 有关 switch 和 ? : 的测试条件及其他模式的更多信息，参见本系列中的《你不知道的 JavaScript（中卷）》第一部分。

2.4 严格模式

ES5 为这个语言新增了“严格模式”，严格限制了某些行为的规则。一般来说，这些限制可以将代码保持在一个更安全、更适当的规范集合之内。另外，遵循严格模式也更容易让引擎优化你的代码。严格模式是代码的一次重大突破，你应该在自己的程序中一直使用。

根据严格模式编译指示放置的位置，你可以选择使用单独的函数或者整个文件来遵循严格模式：

```
function foo() {
    "use strict";

    // 这个代码是严格模式

    function bar() {
        // 这个代码是严格模式
    }
}
// 这个代码不是严格模式
```

对比：

```
"use strict";

function foo() {
```

```
        // 这个代码是严格模式

        function bar() {
            // 这个代码是严格模式
        }
    }

    // 这个代码是严格模式
```

使用严格模式的一个关键区别（改进！）是，不允许省略 var 的隐式自动全局变量声明：

```
function foo() {
    "use strict";    // 开启严格模式
    a = 1;           // 省略var，出现ReferenceError错误
}

foo();
```

如果你在代码中打开严格模式，但代码报错或者开始出现 bug，这可能会诱使你避开严格模式。但这个本能是一个坏习惯。如果严格模式导致程序出现问题，几乎可以确定这标志着你的程序中有些东西应该进行修复。

严格模式不只会让你的代码更加安全或者更易于优化，更代表了这门语言未来的发展方向。现在就要开始习惯严格模式，而不是一直往后推，这对你来说更简单一些，因为转变更晚只会更难！

> 有关严格模式的更多信息，参见本系列《你不知道的 JavaScript（中卷）》第一部分的第 5 章。

2.5 作为值的函数

到目前为止，我们已经介绍了 JavaScript 中作为主要作用域机制的函数。回忆一下典型的函数声明语法是怎样的：

```
function foo() {
    // ..
}
```

虽然从这个语法上看可能不是很明显，但 foo 基本上就是一个外层作用域中的一个变量，这个作用域赋予被声明函数一个引用。也就是说，这个函数本身是一个值，就像 42 或者 [1,2,3] 一样。

这个概念乍听起来可能很奇怪，你需要花点时间来理解它。不仅你可以向函数传入值（参数），**函数本身也可以作为值**赋给变量或者向其他函数传入，又或者从其他函数传出。

因此，应该将函数值视为一个表达式，与其他的值或者表达式类似。

考虑：

```
var foo = function() {
    // ..
};

var x = function bar(){
    // ..
};
```

第一个赋给变量 `foo` 的函数表达式被称为是**匿名的**，因为这个函数表达式没有名称。

第二个函数表达式是已命名的（`bar`），即使它的引用赋值给了变量 `x`。虽然**匿名函数表达式**的使用仍然极为广泛，但通常更需要**已命名函数表达式**。

要想获取更多信息，参见本系列中的《你不知道的 JavaScript（上卷）》第一部分。

2.5.1　立即调用函数表达式

在前面的代码片段中，两个函数表达式都没有运行——如果加上 `foo()` 或者 `x()`，那么就可以执行了。

还有另一种方法可以执行函数表达式，这种方法通常被称为**立即调用函数表达式**（immediately invoked function expression，IIFE）：

```
(function IIFE(){
    console.log( "Hello!" );
})();
// "Hello!"
```

`(function IIFE(){ .. })` 函数表达式外面的 `( .. )` 就是 JavaScript 语法能够防止其成为普通函数声明的部分。

表达式最后的 `()`（即 `})();` 这一行）实际上就表示立即执行前面给出的函数表达式。

这看起来可能有点奇怪，但实际上并不像初看上去那么诡异。思考 `foo` 和这里的 `IIFE` 的类似之处：

```
function foo() { .. }

// foo函数引用表达式，然后()执行它
foo();

// IIFE函数表达式，然后()执行它
(function IIFE(){ .. })();
```

正如你看到的，在运行 () 前列出 (function IIFE(){ .. }) 本质上和执行 () 之前的 foo 是一样的；两种情况都是使用 () 执行了在它之前的函数引用。

因为 IIFE 就是一个函数，而且函数会创建新的变量**作用域**，所以使用 IIFE 的这种风格也常用于声明不会影响 IIFE 外代码的变量：

```
var a = 42;

(function IIFE(){
    var a = 10;
    console.log( a );    // 10
})();

console.log( a );        // 42
```

IIFE 也可以有返回值：

```
var x = (function IIFE(){
    return 42;
})();

x;  // 42
```

以上执行的 IIFE 命名函数返回了值 42，并被赋给了 x。

2.5.2　闭包

闭包是 JavaScript 中一个非常重要，且经常被误解的概念。这里不作深入介绍，可以参见本系列中的《你不知道的 JavaScript（上卷）》第一部分。但我将会解释相关的几个要点，以帮助你理解一般概念。这将会是你 JavaScript 技巧集中最重要的技术之一。

你可以将闭包看作“记忆”并在函数运行完毕后继续访问这个函数作用域（其变量）的一种方法。

考虑：

```
function makeAdder(x) {
    // 参数x是一个内层变量

    // 内层函数add()使用x，所以它外围有一个“闭包”
    function add(y) {
        return y + x;
    };

    return add;
}
```

每次调用外层 makeAdder(..) 返回的、指向内层 add(..) 函数的引用能够记忆传入 makeAdder(..) 的 x 值。现在，我们来使用 makeAdder(..)：

```
// plusOne获得指向内层add(..)的一个引用
// 带有闭包的函数在外层makeAdder(..)的x参数上
var plusOne = makeAdder( 1 );

// plusTen获得指向内层add(..)的一个引用
// 带有闭包的函数在外层makeAdder(..)的x参数上
var plusTen = makeAdder( 10 );

plusOne( 3 );       // 4 <-- 1 + 3
plusOne( 41 );      // 42 <-- 1 + 41

plusTen( 13 );      // 23 <-- 10 + 13
```

我们来详细说明一下这段代码是如何执行的。

(1) 调用 `makeAdder(1)` 时得到了内层 `add(..)` 的一个引用，它会将 `x` 记为 `1`。我们将这个函数引用命名为 `plusOne()`。
(2) 调用 `makeAdder(10)` 时得到了内层 `add(..)` 的另一个引用，它会将 `x` 记为 `10`，我们将这个函数引用命名为 `plusTen()`。
(3) 调用 `plusOne(3)` 时，它会向 `1`（记住的 `x`）加上 `3`（内层 `y`），从而得到结果 `4`。
(4) 调用 `plusTen(13)` 时，它会向 `10`（记住的 `x`）加上 `13`（内层 `y`），从而得到结果 `23`。

如果这在刚开始看上去很奇怪，也令人迷惑的话，不要着急！你需要大量实践才能完全理解这个过程。

不过相信我，一旦你理解了，它就会成为所有编程技术中最为强大有用的技术。它绝对值得你花费一些脑力去理解。在下一节中，我们将针对闭包进行更深入的实践。

模块

在 JavaScript 中，闭包最常见的应用是模块模式。模块允许你定义外部不可见的私有实现细节（变量、函数），同时也可以提供允许从外部访问的公开 API。

考虑：

```
function User(){
    var username, password;

    function doLogin(user,pw) {
        username = user;
        password = pw;

        // 执行剩下的登录工作
    }
    var publicAPI = {
        login: doLogin
    };

    return publicAPI;
```

```
}

// 创建一个User模块实例
var fred = User();

fred.login( "fred", "12Battery34!" );
```

函数 User() 用作外层作用域，持有变量 username 和 password，以及内层的函数 doLogin()；这些都是这个 User 模块私有的内部细节，无法从外部访问。



我们有意没有调用 new User()，尽管这事实上可能对多数读者来说更为熟悉。User() 只是一个函数，而不是需要实例化的类，所以只是正常调用就可以了。使用 new 是不合适的，实际上也是浪费资源。

执行 User() 创建了 User 模块的一个**实例**，这创建了一个新的作用域，因而创建了所有内层变量 / 函数的一个新副本。我们将这个实例赋给 fred。如果再次运行 User()，那么会得到一个不同于 fred 的全新实例。

内层的函数 doLogin() 在 username 和 password 上有一个闭包，这意味着即使在 User() 函数运行完毕之后，函数 doLogin() 也保持着对它们的访问权。

publicAPI 是带有一个属性 / 方法 login 的对象，login 是对内层函数 doLogin() 的一个引用。当我们从 User() 返回 publicAPI 时，它就变成了我们命名为 fred 的那个实例。

此时，外层的函数 User() 已经运行完毕。我们通常认为像 username 和 password 这样的内层变量也就随之消失了。但上述示例并不会这样，因为 login() 函数的内部有一个可以使得它们依然保持活跃的闭包。

这就是我们可以调用 fred.login(..) 的原因，这等同于调用内层 doLogin(..)，并且 fred.login(..) 仍然可以访问内层变量 username 和 password。

这里只是对闭包和模块模式的匆匆一瞥，它们的某些细节很可能还是有点令人迷惑。没关系！让你的大脑完全理解它还需要一些努力。

从现在开始，阅读本系列《你不知道的 JavaScript（上卷）》第一部分，这有助于你更进一步地理解这些知识。

2.6 this 标识符

JavaScript 中另一个被普遍误解的概念是 this 关键字。同样地，本系列《你不知道的 JavaScript（上卷）》第二部分中有几章内容是专门介绍这一技术的，因此，这里只是简单地介绍一下概念。

虽然 this 一般与“面向对象的模式”相关，但 JavaScript 中的 this 则是另外一种机制。

如果一个函数内部有一个 this 引用，那么这个 this 通常指向一个对象。但它指向的是哪个对象要根据这个函数是如何被调用来决定。

this **并不指向**这个函数本身，意识到这一点非常重要，因为这是最常见的误解。

以下是一个简单的说明：

```
function foo() {
    console.log( this.bar );
}

var bar = "global";

var obj1 = {
    bar: "obj1",
    foo: foo
};

var obj2 = {
    bar: "obj2"
};

// --------

foo();              // “全局的”
obj1.foo();         // "obj1"
foo.call( obj2 );   // "obj2"
new foo();          // undefined
```

关于如何设置 this 有 4 条规则，上述代码中的最后 4 行展示了这 4 条规则。

(1) 在非严格模式下，foo() 最后会将 this 设置为全局对象。在严格模式下，这是未定义的行为，在访问 bar 属性时会出错——因此 "global" 是为 this.bar 创建的值。
(2) obj1.foo() 将 this 设置为对象 obj1。
(3) foo.call(obj2) 将 this 设置为对象 obj2。
(4) new foo() 将 this 设置为一个全新的空对象。

底线：为了搞清楚 this 指向什么，你必须检查相关的函数是如何被调用的。调用方式会是以上 4 种之一，这也会回答“this 是什么”这个问题。

> 有关 this 的更多信息，参见本系列《你不知道的 JavaScript（上卷)》第二部分中的前两章。

2.7 原型

JavaScript 中的原型机制是非常复杂的。这里我们仅仅浅谈一下。你需要花费很多时间来阅读本系列《你不知道的 JavaScript（上卷）》第二部分的 4~6 章，以了解所有细节。

当引用对象的某个属性时，如果这个属性并不存在，那么 JavaScript 会自动使用对象的内部原型引用找到另外一个对象来寻找这个属性。你可以将这点看作是属性缺失情况的备用模式。

从一个对象到其后备对象的内部原型引用的链接是在创建对象时发生的。展示这一点的最简单的方法就是使用内置工具 `Object.create(..)`。

考虑：

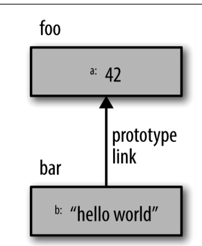
```
var foo = {
    a: 42
};

// 创建bar并将其链接到foo
var bar = Object.create( foo );

bar.b = "hello world";

bar.b;      // "hello world"
bar.a;      // 42 <-- 委托给foo
```

下图直观地展示了对象 `foo`、`bar` 及其相互关系：



对象 `bar` 上的 `a` 属性实际上并不存在，但因为 `bar` 是原型链接到 `foo` 的，所以 JavaScript 自动跳入到 `foo` 对象上搜索 `a` 属性，并且成功找到了。

这种链接看起来像是这个语言的奇怪特性。这个特性最常见的使用（我认为是误用）方式就是模拟 / 伪装带“继承”关系的“类”机制。

更自然应用原型的方式是被称为“行为委托”的模式，其设计意图是，被链接对象能够将其所需要的行为委托给另外一个对象。

有关原型和行为委托的更多信息，参见本系列《你不知道的 JavaScript（上卷)》第二部分的 4~6 章。

2.8 旧与新

在我们已经介绍过的 JavaScript 特性和本系列其余图书将要介绍的更多特性中，有一部分特性是新增的，旧版浏览器不一定会支持这样的特性。实际上，标准中的部分最新特性甚至还没有在哪个稳定版的浏览器中实现。

那么，应该怎样对待这些新特性呢？是不是只能等几年甚至几十年，直到所有这些旧版的浏览器退休呢？

很多人都是这么认为的，但这对 JavaScript 来说实际上是很不健康的一种思路。

你可以使用两种主要的技术，即 polyfilling 和 transpilling，向旧版浏览器“引入”新版的 JavaScript 特性。

2.8.1 polyfilling

单词“polyfill”是由 Remy Sharp 发明的一个新术语（https://remysharp.com/2010/10/08/what-is-a-polyfill），用于表示根据新特性的定义，创建一段与之行为等价但能够在旧的 JavaScript 环境中运行的代码。

举例来说，ES6 定义了一个名为 `Number.isNaN(..)` 的工具，用于提供一个精确无 bug 的 `NaN` 值检查，取代原来的 `isNaN(..)`。但对这个工具进行兼容处理很容易，这样一来，无论终端用户是否使用 ES6 浏览器，你都能够开始使用它。

考虑：

```
if (!Number.isNaN) {
    Number.isNaN = function isNaN(x) {
        return x !== x;
    };
}
```

`if` 语句防止在已经支持此特性的 ES6 浏览器中应用 polyfill 定义。如果还不存在的话，那我们就定义 `Number.isNaN(..)`。

我们进行的这个检查利用了 `NaN` 值的一个特性，即 `NaN` 是整个语言中唯一和自身不相等的值。因此，`NaN` 是使得 `x != x` 为真的唯一值。

并非所有的新特性都是可以兼容旧环境的。有时一个特性的绝大部分可以兼容旧环境，但仍有微小的偏离。如果你要亲自进行 polyfilling 的话，一定要特别小心，确保尽可能严格地遵循标准规定。

或许更好的办法是，使用一个已有的、可信任的 polyfilling 版本，比如由 ES5-Shim（https://github.com/es-shims/es5-shim）和 ES6-Shim（https://github.com/es-shims/es6-shim）提供的版本。

2.8.2 transpiling

语言中新增的语法是无法进行 polyfilling 的。新语法在旧版 JavaScript 引擎上会抛出未识别 / 无效错误。

因此，更好的方法是，通过工具将新版代码转换为等价的旧版代码。这个过程通常被称为“transpiling”。它是由 transforming（转换）和 compiling（编译）组合而成的术语。

从本质上来说，你的源码是用新语法形式编写的，但部署在浏览器上的是编译转换后的旧语法形式。通常会在构建过程中插入 transpiler 工具，类似于代码 linter 或者 minifier。

你可能会疑惑为什么要这么麻烦地编写新语法代码，难道只是为了将它编译转换到旧版代码——为什么不直接编写旧语法代码呢？

有几点重要原因使得 transpiling 值得被关注。

- 语言中新添加的语法的设计目的是让代码更容易阅读和维护。等价的旧版本通常更加繁复。你应该编写更新、更简洁的语法，这不只是为你自己，同时也是为开发组中的所有其他成员着想。
- 如果只是为旧版本进行编译转换，对新版本应用新语法，那么你就得到了新语法浏览器性能优化的好处。这也使得浏览器开发者可以拥有更真实的代码，以便测试它们的实现和优化。
- 越早使用新语法，就可以越早在现实世界中更健壮地测试这些语法，也就可以越早地为 JavaScript 委员会（TC39）提供反馈。如果能够很早就发现问题，那么就能够在这些语言设计错误被固化前对其进行修改 / 修复。

以下是 transpiling 的一个简单示例。ES6 新增了一个名为“默认参数值”的新特性。如下所示：

```
function foo(a = 2) {
    console.log( a );
}

foo();      // 2
foo( 42 );  // 42
```

很简单，对不对？但也非常有用！然而这个新语法在 ES6 前的引擎中是无效的。那么

transpiler 是如何改变这段代码，从而让其能够在旧环境下运行的呢？

```
function foo() {
    var a = arguments[0] !== (void 0) ? arguments[0] : 2;
    console.log( a );
}
```

正如你可以看到的，它会检查 arguments[0] 的值是否为 void 0（也就是 undefined），如果是的话就提供默认值 2；否则就使用传入值。

除了能够在旧版浏览器中使用更好的新语法，编译转换后的代码实际上也更好地表达了编程意图。

单看这段 ES6 版本的代码，你可能不会意识到 undefined 是唯一一个无法作为默认值参数显式传入的值。而编译转换后的代码就更清楚地展示了这一点。

关于 transpiler 最后要强调的重要细节是，现在应该将它看作是 JavaScript 开发生态环境和过程的一个标准部分。JavaScript 将会持续进化，比以往更快，所以每隔几个月就会添加新的语法和特性。

如果你默认使用 transpiler，只要发现新的语法有用就能够一直转换到新语法，而无需等到多年以后当前浏览器被淘汰。

有很多很棒的 transpiler 可供选择。以下是编写本部分时几个很好的选择：

- Babel（https://babeljs.io/，从 6 到 5）
  从 ES6+ 编译转换到 ES5

- Traceur（https://github.com/google/traceur-compiler）
  将 ES6、ES7 及后续版本转换到 ES5

2.9 非 JavaScript

到目前为止，我们介绍的内容都局限于 JavaScript 语言本身。而现实情况是，大多数的 JavaScript 都是编写用于在浏览器这样的环境中运行并与之交互的。严格来说，你编写的代码很大一部分并不直接由 JavaScript 控制。这听起来有点奇怪。

你将遇到的最常见的非 JavaScript 就是 DOM API。举例来说：

```
var el = document.getElementById( "foo" );
```

当你的代码在浏览器中运行时，变量 document 作为一个全局变量存在。它既不是由 JavaScript 引擎提供的，也不由 JavaScript 标准控制。它的存在形式看起来非常类似于普通的 JavaScript 对象，但实际上并不完全是这样。它是一个特殊的对象，通常被称为“宿

主对象”。

另外，`document` 上的方法 `getElementById(..)` 看起来像是一个正常的 JavaScript 函数，但它其实是浏览器的 DOM 提供的指向内置方法的一个很薄的暴露接口。在某些（新版的）浏览器中，这一层可能在 JavaScript 中，但传统的 DOM 及其行为更可能是用 C/C++ 实现的。

另一个示例是输入 / 输出 (I/O)。

广受喜爱的 `alert(..)` 会在用户浏览器窗口弹出一个消息框。`alert(..)` 是由浏览器提供给 JavaScript 程序的，而不是由 JavaScript 引擎本身提供。你发起的调用将消息发送到浏览器内部，然后由它负责绘制并显示消息框。

`console.log(..)` 也是如此；你的浏览器提供了这样的机制并将其连接到开发者工具中。

本书以及本系列主要关注 JavaScript 语言。因此你不会看到这些非 JavaScript 的机制。但你需要了解这些知识，因为你编写的每个 JavaScript 程序都需要用到它们！

2.10　小结

学习使用 JavaScript 编程的第一步就是要了解其核心机制，比如值、类型、函数闭包、`this` 以及原型。

当然，你在本部分看到的每个主题都值得更深入的学习，这也是本系列其他部分专门讲述这些概念的原因。充分理解本章中的概念和代码示例后，本系列的其他部分就等着你去真正挖掘了，希望你能对这门语言获得更深入的理解。

本部分的最后一章简单总结了本系列其他部分的主题，以及那些我们还没有讨论过的概念。

第 3 章

深入“你不知道的 JavaScript”系列

这个系列到底讲了什么？简单地说，这个系列视学习**全部的** JavaScript 为一个严肃的任务，不仅仅是这门语言中被称为“精髓”的某个子集，也不是你完成工作所需要的最小集合。

学习其他语言的认真的开发者会想要花费精力学习他们使用的主要语言的方方面面，但 JavaScript 开发者却通常不会学习这个语言的很多内容，从这个意义上来说，他们似乎是特立独行的。这并不是好事情，也不是我们应该继续放任其发展的事情。

“你不知道的 JavaScript”系列与普遍的 JavaScript 学习方法形成鲜明对比，几乎与任何其他你能读到的 JavaScript 相关书籍都有所不同。它能够让你超越自己的舒适区，对遇到的每个特性提出更深层次的“为什么”。你准备好迎接挑战了吗？

我将在本章中简单总结一下本系列其余几本图书的内容，以及如何基于这个系列最有效地构建 JavaScript 学习的基础。

3.1 作用域和闭包

参见《你不知道的 JavaScript（上卷）》第一部分。

变量的作用域到底是如何在 JavaScript 中工作的？这可能是你需要快速理解的一个最基础的事情了。对作用域只有**道听途说、模糊不清**的理解是不够的。

“作用域和闭包”这部分从批判 JavaScript 是“解释性语言”因而无法编译这一常见误解开始。事实并非如此。

JavaScript 引擎在执行前（有时是执行中！）就编译了代码。因此，通过深入理解编译器对代码的处理方式，我们可以尝试理解它是如何找到并处理变量和函数声明的。沿着这条路，我们看到了 JavaScript 变量作用域管理的常见方式——“提升”。

对“词法作用域”的关键理解是我们在上卷第一部分最后一章继续研究闭包的基础。闭包可能是 JavaScript 所有概念中最重要的一个，但如果你没有深刻了解作用域的工作原理，那么很可能就无法理解闭包。

正如我们在第 2 章中简单提到的那样，闭包的一个重要应用就是模块模式。模块模式可能是 JavaScript 所有代码组织模式中最普遍的方法；深入理解模块模式应该是你最高优先级的任务之一。

3.2 this 和对象原型

参见《你不知道的 JavaScript（上卷)》第二部分。

有关 JavaScript 流传最广、最持久的不实论点是，关键词 `this` 指向它所在的函数。这简直错得离谱。

关键词 `this` 是根据相关函数的执行方式而动态绑定的，事实证明，可以通过 4 条简单的规则理解并完全确定 `this` 绑定。

与 `this` 紧密关联的是对象原型机制，这种机制是一个属性查找链，与寻找词法作用域变量的方式类似。但在原型中进行封装，即模拟（伪造）类和（所谓“原型化的”）继承，是对 JavaScript 的另一个重大误用。

不幸的是，将类和继承的设计模式思维带入 JavaScript 的想法是你所做的最坏的事情，因为语法可能会让你迷惑不已，让你以为真的有类这样的东西存在，实际上原型机制与类的行为特性是完全相反的。

问题是，忽略这种不一致性而假装你实现的就是“继承”更好，还是学习和接受对象原型系统真实的工作方式更有益呢？后者被更合理地命名为“行为委托”。

这不只是语法偏好的问题。委托是完全不同的设计模式，也更加强大，它取代了需要类和继承的设计。但是这些判断违背了这个主题在 JavaScript 的整个生命周期的每个博文、图书和会议发言中的说法。

我对委托与继承的看法并非出自对这门语言及其语法的厌恶，而是来自对使用这个语言真实能力的期待，以及消除无休止的迷惑和沮丧的期待。

但我就原型和委托给出的解释示例要比这里的内容深入许多。如果你已经准备好重新思考对 JavaScript 的“类”和“继承”的所有认知，那么我为你提供“吃下红色药丸”（《黑客

帝国》，1999）的机会，阅读“this 和对象原型”这部分的第 4~6 章吧。

3.3 类型和语法

参见《你不知道的 JavaScript（中卷）》第一部分。

“类型和语法”这部分主要关注另一个高度争议的主题：强制类型转换。当讨论有关隐式类型转换的迷惑时，没有比这个主题更令 JavaScript 开发者烦恼的了。

到目前为止，传统的认知是，隐式类型转换是这个语言中“坏的部分”，应该不惜代价地予以避免。实际上，有些人甚至称其为这个语言的设计“缺陷”。确实，一些工具所做的所有事情就是搜索代码，抱怨你是否进行了类型转换这样的事情。

但是，类型转换是否真的这么令人迷惑、这么坏、这么危险，以至于你一使用就会毁灭自己的代码呢？

我认为并非如此。在第 1~3 章理解了类型和值到底是如何工作的后，第 4 章讨论了这个争议，并完整地解释了类型转换是如何工作的，包括所有的边边角角。

我们会看到，到底类型转换的哪些部分是出乎意料的，哪些部分在花费精力学习后则是完全可以理解的。

这不仅仅只是声称类型转换是合理的、可学习的；我想表明的是，类型转换是非常有用且被低估了的工具，**你应该在自己的代码中使用它**。在我看来，如果能够正确使用的话，类型转换不仅能够工作，而且也会让你的代码质量更高。所有的反对者和怀疑者肯定会嘲笑这样的立场，但我坚信这是提高你 JavaScript 水平的关键一点。

你只想人云亦云、随波逐流吗？还是你愿意将所有的假设放到一边，用全新的视角来观察类型转换？“类型与语法”部分将会转换你的思路。

3.4 异步和性能

参见《你不知道的 JavaScript（中卷）》第二部分。

“作用域和闭包”“`this` 和对象原型”以及“类型和语法”关注的都是语言的核心机制，而“异步和性能”则稍微偏重于在语言机制之上处理异步编程的模式。异步不只是对应用的性能至关重要，而且正在慢慢成为代码易写性和可维护性方面的关键因素。

“异步和性能”部分一开始明确了大量的术语和概念，如“异步”“并行”和“并发”这些概念，并深入解释了这些概念为什么适用或不适用于 JavaScript。

然后我们查看了回调这个使得异步成为可能的基本方法。但我们很快就看到，单独使用回

调完全不足以满足当代异步编程的需求。我们确定了两种只用回调编码的缺陷：**控制反转**（Inversion of Control，IoC）信任缺失和线性理解能力的缺失。

为了避免这两个主要的缺陷，ES6 引入了新的机制（实际上是模式）：promise 和生成器。

promise 是对“未来值”的与时间无关的封装，使得不管这个值是否已经可用，你都可以推导和组合使用它们。另外，通过一种可信任的、可组合的 promise 机制，分发回调它们也有效地解决了 IoC 信任问题。

生成器为 JavaScript 函数引入了一种新的执行模式，其中生成器可以暂停在 `yield` 点上，并在之后被异步继续。暂停与继续的能力使得生成器中同步的、看似连续的代码可以在后台异步执行。通过这种方式，我们解决了回调的非线性、非局部跳转引发的代码混乱问题，因而让我们的异步代码看似同步，更容易追踪。

但是，promise 和生成器的组合“暂缓”了我们最有效的异步编码模式进入 JavaScript 的日程。实际上，ES7 及更新版本中即将出现的异步的高级机制很大程度上是建立在这个基础上的。要想在异步的世界中严肃地对待程序效率，你需要非常熟悉 promise 和生成器的组合。

如果说 promise 和生成器与表达模式有关，这种模式使得我们的程序可以更加并发地运行，因此能在更短的时间内处理完毕，那么 JavaScript 还有很多其他的性能优化因素值得探讨。

第 5 章探讨了通过 Web Worker 实现程序并行和通过 SIMD 实现数据并行的主题，以及像 ASM.js 这样的底层优化技术。第 6 章从合适的测评技术角度介绍了性能优化，其中包括哪些类型的性能需要关注，哪些可以忽略。

编写高效的 JavaScript 代码意味着，你编写的代码可以打破不同浏览器和环境的壁垒，达到动态运行。这要求大量复杂而详细的计划和努力，只有这样，才能让程序从“可以运行”到“可以很好地运行”。

“异步和性能”部分的目的是，为你提供编写合理、高性能 JavaScript 代码所需要的所有工具和技巧。

3.5 ES6 及更新版本

参见本书第二部分。

不管你认为自己此时对 JavaScript 已经有了怎样的掌握，事实是 JavaScript 一直在持续发展，而且，发展的速度越来越快。这个事实几乎就是本系列精神的隐喻，你需要接受我们永远无法完全了解 JavaScript 这个事实，因为即使你掌握了所有内容，还是会出现你需要学习的新东西。

这个主题为这门语言的发展方向提供了短期和中期的展望，不只是 ES6 这样已知的方向，还有在其之后可能的方向。

本系列中的所有内容都是根据撰写时的 JavaScript 的状态编写的，即 ES6 的接受期。本系列的关注点更多在 ES5 上。现在，我们要将注意力放在 ES6、ES7 及更远的未来……

既然编写本系列时 ES6 已经接近完成，“ES6 及更新版本”部分一开始就将 ES6 中的具体内容分成了几个关键的类别，其中包括新语法、新数据结构（集合）以及新处理能力和 API。我们将介绍 ES6 的每一个新特性，详细程度有所不同，并且还会回顾本系列其他部分提到的细节。

预先列出令人兴奋的 ES6 特性：解构、默认参数值、符号、简洁方法、计算属性、箭头函数、块作用域、promise、生成器、迭代器、模块、代理、WeakMap，以及更多！啊，ES6 确实改进了不少！

“ES6 及更新版本”的前面是一个线路图，可以帮助你了解改进后的 JavaScript，这可是你在未来几年里需要编写和探索的 JavaScript。其后面简单关注了我们有把握在 JavaScript 的近期可期待的新特性。最重要的实现就是后 ES6，JavaScript 很可能是一个特性一个特性地演化，而不是一个版本一个版本地演化，这意味着这些近期将要出现的特性可能会比你想象的更快到来。

JavaScript 的未来是光明的。是时候开始学习了吧！

3.6 小结

“你不知道的 JavaScript”系列的宗旨是，所有的 JavaScript 开发者都可以，也应该学习这门伟大语言的方方面面。个人偏见、框架假定和项目的截止日期都不应该成为你从不学习和深入理解 JavaScript 的借口。

我们覆盖了这门语言中每个重要的焦点领域，完整地探索了所有你可能以为自己已经了解，却并没有完全理解的部分。

“你不知道的 JavaScript”既不是批判也不是攻击。这是一种领悟，是包括我自己在内的所有人都必须接受的事实。学习 JavaScript 不是最终目标，而是一个过程。我们还不了解 JavaScript。但我们终将做到这一点！

第二部分

ES6 及更新版本

序

Kyle Simpson 是一位缜密的务实主义者。

这是我能想到的最高赞美。对我来说，这是软件开发者必须具备的最重要的品质。对，是必须而不是应该。把 JavaScript 编程语言的各个层次梳理清楚，并以含义丰富且易于理解的形式呈现出来，Kyle 在这方面的敏锐能力是首屈一指的。

“你不知道的 JavaScript”系列的读者将会了解“ ES6 及更新版本”，从最直观的到那些我们一直认为理所当然或者从未想到的微妙语义，他们将会沉浸于其间。到目前为止，“你不知道的 JavaScript”系列图书覆盖的内容是读者至少在某种程度上已经熟悉的，看到或者听说过这些主题，甚至可能实践过。 而本书这一部分将要介绍的内容只有 JavaScript 开发者社区的很少一部分人有所了解，即 ECMAScript 2015 语言规范的发展变化。

过去几年里，我看到了 Kyle 不懈努力地学习熟悉这部分内容，最终达到了只有少数专业人士所能精通的程度。在他写作这部分内容时，ECMAScript 2015 语言规范文档还没有正式发布，能考虑到这一点是多么地了不起！我所说的千真万确，我阅读了 Kyle 为这一内容所写的每一个字。我还跟踪了他的所有修改，每一次修改都是对内容的改进，为我们提供了更深层次的认识。

这些内容展示了未知的新知识，以此来触发读者的思考，使你的知识储备与工具一起进化。它还会为读者增加信心来彻底拥抱 JavaScript 编程的下一个重要时代。

——Rick Waldron (@rwaldron)，Bocoup 公司开源 Web 开发者，Ecma/TC39 jQuery 代表

第 1 章

ES? 现在与未来

在深入阅读本部分之前，你应该已经能够熟练应用（直到编写本部分时）最新标准下的 JavaScript 工作了，这个标准通常被称为 ES5（严格说是 ES5.1）。本部分中，我们将会直接介绍 ES6，同时也会扩展视野，理解 JavaScript 未来的发展方向。

如果你对自己的 JavaScript 水平还不是那么有信心的话，我强烈建议你先阅读一下本系列《你不知道的 JavaScript（上卷）》和《你不知道的 JavaScript（中卷）》中的几部分内容。

- 作用域和闭包：你知道 JavaScript 的词法作用域是基于编译器（而非解释器！）语义的吗？你能解释词法作用域和作为值的函数这两者的直接结果之一就是闭包吗？
- `this` 和对象原型：你能复述 `this` 绑定的四条基本原则吗？你是否还在用 JavaScript 的“伪”类应付了事，而没有采用更简洁的“行为委托”设计模式？你听说过**连接到其他对象的对象**（objects linked to other objects，OLOO）吗？
- 类型和语法：你了解 JavaScript 中的内置类型吗？更重要的是，你了解如何正确安全地使用类型间强制转换吗？对于 JavaScript 语法 / 句法中的微妙细节，你的熟悉程度又如何？
- 异步和性能：你还在使用回调管理异步吗？你能解释 promise 是什么以及它为什么 / 如何能够解决“回调地狱”这个问题吗？你知道如何应用生成器来使得异步代码更加清晰吗？对 JavaScript 程序和具体运算的深度优化到底由哪些方面构成？
- 起步上路：你还是编程新手或者 JavaScript 新手吗？这是你起步之时需要了解的技术发展路线（本书第一部分）。

如果你阅读了前面所有内容，并对其覆盖的主题有了足够的信心，现在是我们深入探索 JavaScript 已经到来的及更远未来将会出现的改变的时候了。

与 ES5 不同，ES6 并不仅仅是为这个语言新增一组 API。它包括一组新的语法形式，其中的一部分可能是要花些时间才能理解和熟悉的。它还包括各种各样的新的组织形式和操作各种数据类型的新的辅助 API。

对于这个语言来说，ES6 是一次激进的飞跃。即使你认为自己对 JavaScript ES5 已经相当了解，ES6 还是有大量你不知道的全新内容。所以做好准备吧！这里探讨了你需要了解的 ES6 的所有重要主题，并且简单介绍了你可能还没有意识到的未来将会出现的特性。

> 本部分中的所有代码假定运行环境为 ES6+。在编写本部分的时候，各种浏览器和 JavaScript 环境（比如 Node.js）对 ES6 的支持程度差异巨大，所以运行结果可能也会有所差异。

1.1 版本

JavaScript 标准的官方名称是“ECMAScript”（简称“ES”），直到最近都是用有序数字来标识版本的，例如“5”表示“第 5 版”。

最早的 JavaScript 版本是 ES1 和 ES2，它们不怎么为人所知，实现也很少。第一个流行起来的 JavaScript 版本是 ES3，它成为浏览器 IE6-8 和早前的旧版 Android 2.x 移动浏览器的 JavaScript 标准。出于某些政治原因，倒霉的 ES4 从来没有成形，这里我们不做讨论。

2009 年，ES5 正式发布（然后是 2011 年的 ES5.1），在当代浏览器（包括 Firefox、Chrome、Opera、Safari 以及许多其他类型）的进化和爆发中成为 JavaScript 广泛使用的标准。

下一个 JavaScript 版本（发布日期从 2013 年拖到 2014 年，然后又到 2015 年）标签，之前的共识显然是 ES6。

但是，在 ES6 规范发展后期，出现了这样的方案：有人建议未来的版本应该改成基于年份，比如 ES2016（也就是 ES7）来标示在 2016 年结束之前敲定的任何版本的规范。尽管有异议，但比起后来提出的方案 ES2015，很可能保持统治地位的版本命名仍是 ES6。而 ES2016 可能会采用新的基于年份的命名方案。

人们已经观察到，JavaScript 的发展步伐要比每年一个版本快得多。一旦一个想法进入标准讨论阶段，浏览器就会开始为这个特性开发原型，早期的使用者也就会开始编码进行试验了。

对于一个特性来说，通常在官方正式批准之前很久，通过早期的引擎 / 工具原型，这个特性就已经事实上标准化了。所以，把 JavaScript 未来的版本看成基于单个特性，而不是基于某一组主要特性组合为单位（就像现在）的新版本，甚至是每年一个版本（就像以后采用的形式），也是合理的。

这带来的结果是，版本标签已经不再那么重要，JavaScript 开始被看作更像是一个发展的动态的标准。对待这一结果最好的办法是别再把你的代码看作是基于版本的，比如“基于 ES6”，而是去考虑它具体支持哪些单独的特性。

1.2 transpiling

特性演变迅速也让 JavaScript 开发者遇到了问题，他们非常想要使用新特性，但同时又被现实所困，他们的网站 /app 可能需要支持并未提供这些新特性的旧版浏览器，而功能特性的快速进化使得这个问题更加严峻。

ES5 广泛应用于工业界的过程中，其典型思路是要等到绝大多数——如果不说所有——前 ES5 环境都不再需要被支持的时候，代码才采用 ES5。结果是，很多实现才开始（在编写本部分的时候）使用像 `strict` 模式这样的特性，而这早在五年以前就已经出现在 ES5 中了。

人们普遍认为，落后规范这么多年对于 JavaScript 生态系统的未来是有害的。所有负责语言发展的人士都希望，新的特性和模式一旦在标准中稳定下来，并且浏览器能够实现它们之后，就能够在开发者的代码中得到应用。

所以我们如何解决这个矛盾呢？答案是工具化（提供工具）。具体来说是一种称为 transpiling（transformation + compiling，转换 + 编译）的技术。简单地说，其思路是利用专门的工具把你的 ES6 代码转化为等价（或近似！）的可以在 ES5 环境下工作的代码。

举例来说，考虑短路属性定义（参见 2.6 节）。下面是 ES6 形式：

```
var foo = [1,2,3];

var obj = {
    foo     // 也就是foo: foo
};

obj.foo;    // [1,2,3]
```

而下面是转化后的（大致）代码：

```
var foo = [1,2,3];

var obj = {
    foo: foo
};

obj.foo;    // [1,2,3]
```

这个变换很小但能给人带来方便，使我们在同名的时候，可以把对象字面声明的 `foo: foo` 简写为 `foo`。

通常在构建过程中使用 transpiler 执行这些转换，如同执行 linting、minification，或者其他类似操作的步骤。

shim/polyfill

并非所有的 ES6 新特性都需要使用 transpiler，还有 polyfill（也称为 shim）这种模式。在可能的情况下，polyfill 会为新环境中的行为定义在旧环境中的等价行为。语法不能 polyfill，而 API 通常可以。

举例来说，`Object.is(..)` 是一个用于检查两个值严格相等的新工具，而且不像 `===` 那样在处理 `NaN` 和 `-0` 值的时候有微妙的例外情况。对 `Object.is(..)` 应用 polyfill 非常简单：

```
if (!Object.is) {
    Object.is = function(v1, v2) {
        // 检查-0
        if (v1 === 0 && v2 === 0) {
            return 1 / v1 === 1 / v2;
        }
        // 检查NaN
        if (v1 !== v1) {
            return v2 !== v2;
        }
        // 其余所有情况
        return v1 === v2;
    };
}
```

> 注意这个 polyfill 外层用于保护的 `if` 语句。这个细节很重要，它表示这段代码只定义了在未定义 API 的旧环境下的行为；需要覆盖已经存在的 API 的情况是非常罕见的。

这里有一组名为"ES6 Shim"的 ES6 shim 实现（https:// github.com/paulmillr/es6-shim/），你一定要把它作为一个标准放在你所有的 JavaScript 新项目中。

人们认为 JavaScript 会持续不断地发展，浏览器会逐渐地而不是以大规模突变的形式支持新特性。所以，保持 JavaScript 发展更新的最好战略就是在你的代码中引入 polyfill shim，并且在构建过程中加入 transpiler 步骤，现在就开始接受并习惯这个新现实吧。

如果还要保持现状，等着所有浏览器都支持某个特性才开始应用这个特性，那么你就已经落后了。你将遗憾地错过所有设计用于使得编写 JavaSript 更高效健壮的创新。

1.3 小结

在编写本部分时，ES6（有些人可能想要称其为 ES2015）刚刚出现，其中有很多新的技术需要学习！

但是更重要的是，你要转变你的思路以符合 JavaScript 新的发展方式。不要再像过去很多人所做的那样，等待多年直到某个正式文档投票通过（才开始应用这个新特性）。

现在，JavaScript 的新特性一旦可行就会进入浏览器，因此是早早步入正轨还是多年以后再来追赶由你自己决定。

不管未来的 JavaScript 采用何种版本标识，它的发展都会比过去要快得多。transpiler 和 shim/polyfill 都是重要的工具，它们能帮助你保持处于这个语言发展的最前沿。

如果说 JavaScript 发展的新图景中有任何要点需要了解的话，就是现在所有的 JavaScript 开发者都被强烈要求从追踪发展的状态转换为直接站在最前沿。那么，学习 ES6 就是开始的第一步！

第 2 章

语法

有一点比较奇怪的是，不管你编写 JavaScript 代码的时间有多长，都会觉得语法对你来说非常熟悉。虽然确实有一些怪异的地方，但总的来说，JavaScript 语法非常合理自然，它借鉴于其他一些语言，和它们有很多相似之处。

然而，ES6 新增了很多新的语法形式，需要我们去熟悉。本章我们将会介绍这些新的语法形式，看看它们提供了哪些新东西。

> 在编写本部分时，书里讨论的特性中有一些已经被各种浏览器（Firefox、Chrome 等）所支持，但还有一些只是部分实现，其他甚至完全没有实现。直接试验这些示例可能会出现各种结果。如果是这样的话，可以通过 transpiler 来试验，因为这些特性中的绝大多数都已经被此类工具所支持。
>
> ES6Fiddle（http://www.es6fiddle.net/）是一个非常不错的、易于使用的 ES6 试验田，Babel transpiler（http://babeljs.io/repl/）的在线 REPL 也是这样。

2.1　块作用域声明

你很可能已经了解，JavaScript 中变量作用域的基本单元一直是 `function`。如果需要创建一个块作用域，最普遍的方法除了普通的函数声明之外，就是立即调用函数表达式（IIFE）。举例来说：

```
var a = 2;

(function IIFE(){
```

```
        var a = 3;
        console.log( a );   // 3
})();

console.log( a );       // 2
```

2.1.1 let 声明

但现在，我们可以创建绑定到任意块的声明，（不出意外地）其被称为**块作用域**（block scoping）。这意味着我们只需要一对 { .. } 就可以创建一个作用域。不像使用 var 那样声明的变量总是归属于包含函数（即全局，如果在最顶层的话）作用域，而是像下面这样使用 let：

```
var a = 2;

{
    let a = 3;
    console.log( a );   // 3
}

console.log( a );       // 2
```

在 JavaScript 中使用独立的 { .. } 还不是很常见的惯用法，但总是合法的。如果开发者有其他支持块作用域语言的经验，会很容易认出这种模式。

我认为这种使用一个专门的 { .. } 块的模式是创建块作用域变量的最好方法。另外，应该总是把 let 声明放在块的最前面。如果有多个变量需要声明的话，建议只用一个 let。

从编码风格上来说，我甚至愿意把 let 和左括号 { 放在同一行，这样更明确地表明了这个块的目的只是为了声明这些变量的作用域。

```
{   let a = 2, b, c;
    // ..
}
```

这样看起来很奇怪，也不大可能符合其他大多数 ES6 文献中推荐的语法形式。但是我的疯狂是有原因的。

let 声明还有一个试验性（未标准化的）的形式，称为 let 块，看起来就像这样：

```
let (a = 2, b, c) {
    // ..
}
```

我把这种形式称为**显式**块作用域，而镜像于 var 的那种 let .. 声明形式更加**隐式**，因为它某种程度上“劫持”了所在的 { .. } 中的一切。一般来说，开发者喜欢**显式**机制胜过**隐式**机制，我认为这里也是这样。

比较前面的两个代码片段，它们是非常相似的，我认为这两种机制风格上都称得上是**显式**块作用域。不幸的是，其中最显式的 `let (..) { .. }` 形式并没有被 ES6 采用。ES6 之后可能会再次考虑这个选择，但是现在来说，我认为前者是最好的选择。

为了强调说明 `let ..` 声明的**隐式**本质，考虑下面这种用法：

```
let a = 2;

if (a > 1) {
    let b = a * 3;
    console.log( b );       // 6

    for (let i = a; i <= b; i++) {
        let j = i + 10;
        console.log( j );
    }
    // 12 13 14 15 16

    let c = a + b;
    console.log( c );       // 8
}
```

不要回头去看前面的代码，快速回答：哪个（些）变量只存在于 `if` 语句内部，哪个（些）变量只存在于 `for` 循环内部？

答案：`if` 语句包含了块作用域变量 `b` 和 `c`，块作用域变量 `i` 和 `j` 存在于 `for` 循环之中。

你是否需要思考一会呢？ `i` 并不在包含它的 `if` 语句作用域中，这一点是否让你吃惊呢？这种疑惑和思考，我称之为"脑力税"，来自于 `let` 机制对于我们来说不只是新的，同时也是隐式的这个事实。

作用域内部后面才出现的 `let c = ..` 声明也有隐患。和传统的 `var` 声明变量不同，不管出现在什么位置，`var` 都是归属于包含它的整个函数作用域。`let` 声明归属于块作用域，但是直到在块中出现才会被初始化。

在 `let` 声明 / 初始化之前访问 `let` 声明的变量会导致错误，而使用 `var` 的话这个顺序是无关紧要的（除了代码风格方面）。

考虑：

```
{
    console.log( a ); // undefined
    console.log( b ); // ReferenceError!

    var a;
    let b;
}
```

过早访问 let 声明的引用导致的这个 ReferenceError 严格说叫作**临时死亡区**（Temporal Dead Zone，TDZ）错误——你在访问一个已经声明但没有初始化的变量。以后我们还会遇到 TDZ 错误——ES6 中它出现了多次。另外，注意“初始化”并不要求代码中的显式赋值，比如 let b；这样完全是有效的。声明时没有赋值的变量会自动赋值为 undefined，所以 let b; 就等价于 let b = undefined;。不管是否显式赋值，都不能在 let b 语句运行之前访问 b。

最后一点：对于 TDZ 值和未声明值（以及声明过的！），typeof 结果是不同的。举例来说：

```
{
    // a未声明
    if (typeof a === "undefined") {
        console.log( "cool" );
    }

    // b声明了，但还处于TDZ
    if (typeof b === "undefined") {    // ReferenceError!
        // ..
    }

    // ..

    let b;
}
```

这里 a 是未声明的，所以 typeof 是检查它是否存在的唯一安全的方法。而 typeof b 会抛出 TDZ 错误，因为代码中在后面恰好有一个 let b 声明。

现在能更清楚为什么我坚持认为应该把所有的 let 声明放在其所在作用域的最前面了吧。这样就完全避免了不小心过早访问的问题。这也使得阅读这个块，以及所有块代码开头的时候能够更清楚地了解这块代码包含了哪些变量。

你的代码块（if 语句、while 循环等）不需要与块行为方式共享原来的行为方式。

完全由你来选择是否遵守这个规则来提高你的代码的明晰性，而这一点将会节省你大量的重构精力，避免自作自受。

关于 let 和块作用域的更多信息，参见本系列《你不知道的 JavaScript（上卷）》第一部分的第 3 章。

let +for

我建议使用 let 声明块的显式形式，唯一的例外是 let 出现在 for 循环头部的情况。原因可能有点微妙，但是我认为这是 ES6 的重要特性之一。

考虑：

```
var funcs = [];

for (let i = 0; i < 5; i++) {
    funcs.push( function(){
        console.log( i );
    } );
}

funcs[3]();    // 3
```

`for` 循环头部的 `let i` 不只为 `for` 循环本身声明了一个 `i`，而是为循环的每一次迭代都重新声明了一个新的 `i`。这意味着 loop 迭代内部创建的闭包封闭的是每次迭代中的变量，就像期望的那样。

如果试验同样的代码，只把 `var i` 放在 `for` 循环头部，得到的结果就会是 `5` 而不是 `3`，因为在外层作用域中只有一个 `i`，这个 `i` 被封闭进去，而不是每个迭代的函数会封闭一个新的 `i`。

还可以通过下面更详细一点的代码完成同样的事情：

```
var funcs = [];

for (var i = 0; i < 5; i++) {
    let j = i;
    funcs.push( function(){
        console.log( j );
    } );
}

funcs[3]();    // 3
```

这里我们强制在每个迭代内部创建了一个新的 `j`，然后闭包的工作方式是一样的。我更喜欢前面一种形式；这个特殊的额外功能是我为什么支持 `for (let .. ) ..` 形式的原因。可能有人认为这种形式太**隐式**（隐晦），但我认为这已经足够**明晰**，也足够有用了。

`let` 放在 `for .. in` 和 `for .. of` 循环中也是一样的（参见 2.9 节）。

2.1.2　const 声明

还有一个块作用域声明形式需要了解：`const`，用于创建**常量**。

常量到底是什么？它是一个设定了初始值之后就只读的变量。考虑：

```
{
    const a = 2;
    console.log( a );      // 2
```

```
    a = 3;      // TypeError!
}
```

这个变量的值在声明时设定之后就不允许改变。const 声明必须要有显式的初始化。如果需要一个值为 undefined 的**常量**，就要声明 const a = undefined。

常量不是对这个值本身的限制，而是对赋值的那个变量的限制。换句话说，这个值并没有因为 const 被锁定或者不可变，只是赋值本身不可变。如果这个值是复杂值，比如对象或者数组，其内容仍然是可以修改的。

```
{
    const a = [1,2,3];
    a.push( 4 );
    console.log( a );     // [1,2,3,4]

    a = 42;               // TypeError!
}
```

变量 a 并不持有一个常量数组；相反地，它持有一个指向数组的常量引用。数组本身是可以随意改变的。



将一个对象或数组作为常量赋值，意味着这个值在这个常量的词法作用域结束之前不会被垃圾回收，因为指向这个值的引用没有清除。这可能是你想要的，但是如果不想出现这样的情形的话则需要小心。

本质上说，const 声明强化了多年以来我们通过代码风格表达的信号，其中我们把变量名声明为全大写字母，并将其赋值为某个字面值，小心谨慎地不去改变这个值。var 赋值没有什么强制措施，但现在有了 const 赋值，可以帮助我们发现不想出现的改变。

const 可以用在 for、for..in 以及 for..of 循环的变量声明中（参见 2.9 节）。但如果想要重新赋值就会抛出错误，比如 for 循环中常用的 i++。

是否使用 const

有一些传言认为，JavaScript 引擎在某些情况下可以对 const 进行比 let 和 var 更激进的优化。理论上说，引擎更容易了解这个变量的值 / 类型永远不会改变，那么它就可以取消某些可能的追踪。

不管这里 const 是否真的有好处，还是这只是我们自己的幻想和期望，更重要的决定因素为是否需要常量性。记住：源码的一个重要功能是通过清晰的交流表明你的意图，不只是对你自己，也是对未来的你和其他合作者。

有些开发者倾向于一开始就把所有变量都声明为 const，然后如果必须在代码中修改它就改为 let。这个思路很有趣，但是并不确定它是否真正提高代码的可读性和可理解性。

这并不像许多人相信的那样是真正的**保护**，因为任何后续的开发者想要修改一个 `const` 值只需要盲目地把 `const` 声明改成 `let` 即可。最好的情况下，它避免了意外的修改。但是再次说明，除了我们的直觉和感觉，这里没有客观且清晰地说明“意外”是什么或者要防止什么。类似的思路也存在于类型强制转换的情况。

我的建议是：要避免可能令人迷惑的代码，只对你有意表明不会改变的变量使用 `const`。换句话说，不要依赖于 `const` 来规范代码行为，而是在意图清晰的时候，把它作为一个表明意图的工具。

2.1.3　块作用域函数

从 ES6 开始，块内声明的函数，其作用域在这个块内。在 ES6 之前，规范并没有要求这一点，但是许多实现就是这么做的。所以现在是规范与现实保持一致了。

考虑：

```
{
    foo();                      // 可以这么做!

    function foo() {
        // ..
    }
}

foo();                          // ReferenceError
```

`foo()` 函数声明在 `{ .. }` 块内部，ES6 支持块作用域。所以在块外不可用。但是还要注意到它是在块内“提升”了，与 `let` 声明相反，后者会遇到前面介绍的 TDZ 错误陷阱。

如果编写了和前面类似的代码，并依赖于旧有的非块作用域行为，那么函数声明的块作用域就可能会引发问题。

```
if (something) {
    function foo() {
        console.log( "1" );
    }
}
else {
    function foo() {
        console.log( "2" );
    }
}

foo();    // ??
```

在前 ES6 环境中，不管 `something` 的值是什么，`foo()` 都会打印出 `"2"`，因为两个函数声明

都被提升到了块外，第二个总是会胜出。

而在 ES6 中，最后一行会抛出一个 ReferenceError。

2.2 spread/rest

ES6 引入了一个新的运算符 ...，通常称为 spread 或 rest（展开或收集）运算符，取决于它在哪 / 如何使用。我们来看一下：

```
function foo(x,y,z) {
    console.log( x, y, z );
}

foo( ...[1,2,3] );              // 1 2 3
```

当 ... 用在数组之前时（实际上是任何 iterable，我们将在第 3 章中介绍），它会把这个变量“展开”为各个独立的值。

我们通常看到的是前面代码片段中的使用方式，即把一个数组展开为一组函数调用的参数。在这种用法中，... 为我们提供了可以替代 apply(..) 方法的一个简单的语法形式，在前 ES6 中我们常常这样写：

```
foo.apply( null, [1,2,3] );     // 1 2 3
```

然而，... 也可以在其他上下文中用来展开 / 扩展一个值，比如在另一个数组声明中：

```
var a = [2,3,4];
var b = [ 1, ...a, 5 ];

console.log( b );               // [1,2,3,4,5]
```

在这种用法中，... 基本上代替了 concat(..)，这里的行为就像是 [1].concat(a, [5])。

... 的另外一种常见用法基本上可以被看作反向的行为；与把一个值展开不同，... 把一系列值收集到一起成为一个数组。考虑：

```
function foo(x, y, ...z) {
    console.log( x, y, z );
}

foo( 1, 2, 3, 4, 5 );           // 1 2 [3,4,5]
```

在这段代码中，...z 基本上是在说：“把**剩下的**参数（如果有的话）收集到一起组成一个名为 z 的数组。”因为 x 赋值为 1，y 赋值为 2，所以其余的参数 3、4 和 5 被收集到数组 z 中。

当然，如果没有命名参数的话，... 就会收集所有的参数：

```
function foo(...args) {
    console.log( args );
}

foo( 1, 2, 3, 4, 5);          // [1,2,3,4,5]
```

> foo(..) 函数声明中的 ...args 通常称为“rest 参数”，因为这里是在收集其余的参数。我喜欢用“收集”这个词，因为这更好地描述了它的行为而不是它的内容。

这种用法最好的一点是，它为弃用很久的 arguments 数组——实际上它并不是真正的数组，而是类似数组的对象——提供了一个非常可靠的替代形式。因为 args（或者随便你给它起什么名字——很多人喜欢用 r 或 rest）是一个真正的数组，前 ES6 中有很多技巧用来把 arguments 转变为某种我们可以当作数组来使用的东西，现在我们可以摆脱这些愚蠢的技巧了。

考虑：

```
// 按照新的ES6的行为方式实现
function foo(...args) {
    // args已经是一个真正的数组

    // 丢弃args中第一个元素
    args.shift();

    // 把整个args作为参数传给console.log(..)
    console.log( ...args );
 }

// 按照前ES6的老派行为方式实现
function bar() {
    // 把arguments转换为一个真正的数组
    var args = Array.prototype.slice.call( arguments );

    // 在尾端添加几个元素
    args.push( 4, 5 );

    // 过滤掉奇数
    args = args.filter( function(v){
        return v % 2 == 0;
    } );

    // 把整个args作为参数传给foo(..)
    foo.apply( null, args );
}

bar( 0, 1, 2, 3 );                  // 2 4
```

函数 foo(..) 声明中的 ...args 收集参数，console.log(..) 调用中的 ...args 将其展开。

这里很好地展示了运算符 ... 对称而又相反的用法。

除了在函数声明中使用 ...，还有一种情况是 ... 被用于收集值，我们将在 2.5 节中介绍。

2.3 默认参数值

可能 JavaScript 最常见的一个技巧就是关于设定函数参数默认值的。多年以来我们实现这一点的方式是这样的，看起来应该很熟悉：

```
function foo(x,y) {
    x = x || 11;
    y = y || 31;

    console.log( x + y );
}

foo();              // 42
foo( 5, 6 );        // 11
foo( 5 );           // 36
foo( null, 6 );     // 17
```

当然，如果你之前使用过这个模式，就会知道它很有用，但同时又有点危险，比如，如果对于一个参数你需要能够传入被认为是 falsy（假）的值。考虑：

```
foo( 0, 42 );        // 53 <--哎呀，并非42
```

为什么？因为这里 `0` 为假，所以 `x || 11` 结果为 `11`，而不是直接传入的 `0`。

要修正这个问题，有些人会选择增加更多的检查，就像下面这样：

```
function foo(x,y) {
    x = (x !== undefined) ? x : 11;
    y = (y !== undefined) ? y : 31;

    console.log( x + y );
}

foo( 0, 42 );           // 42
foo( undefined, 6 );    // 17
```

当然，这意味着除了 `undefined` 之外的任何值都可以直接传入。然而，`undefined` 会表达“我没有传入信息”这样的信息。除非你确实需要能够传入 `undefined`，它就工作的很好。

这种情况下，可以通过它并不存在于数组 `arguments` 之中来确定这个参数是被省略的，可能就像下面这样：

```
function foo(x,y) {
    x = (0 in arguments) ? x : 11;
    y = (1 in arguments) ? y : 31;

    console.log( x + y );
}

foo( 5 );               // 36
foo( 5, undefined );    // NaN
```

但是如果你不能传递任何值（甚至 `undefined` 也不行）来表明“我省略了这个参数”，那么如何省略第一个参数 `x` 呢？

`foo(,5)` 很吸引人，但这是不合法的语法。`foo.apply(null, [,5])` 看来可以实现这个技巧，但是这里 `apply(..)` 的诡异实现意味着这些参数被当作了 `[undefined,5]`，这当然不是一个省略。

如果继续深入的话，就会发现你只能通过传入比“期望”更少的参数来省略最后的若干参数（例如，右侧的），而无法省略位于参数列表中间或者起始处的参数。

这里应用了一个很重要的需要记住的 JavaScrpt 设计原则：`undefined` 意味着**缺失**。也就是说，`undefined` 和缺失是无法区别的，至少对于函数参数来说是如此。

> 在 JavaScript 的其他一些地方上面提到的设计原则是不成立的，这时候会引起混淆，比如对于有空槽的数组。参见本系列《你不知道的 JavaScript（中卷)》第一部分可以获取更多信息。

了解了所有这些之后，现在我们可以讨论 ES6 新增的一个有用的语法来改进为缺失参数赋默认值的流程。

```
function foo(x = 11, y = 31) {
    console.log( x + y );
}

foo();                  // 42
foo( 5, 6 );            // 11
foo( 0, 42 );           // 42

foo( 5 );               // 36
foo( 5, undefined );    // 36 <-- 丢了undefined
foo( 5, null );         // 5  <-- null被强制转换为0

foo( undefined, 6 );    // 17 <-- 丢了undefined
foo( null, 6 );         // 6  <-- null被强制转换为0
```

注意这些结果，以及它们和前面的方法之间的微妙区别和相似之处。

在函数声明中的 `x = 11` 更像是 `x !== undefined ? x : 11` 而不是常见技巧 `x || 11`，所以在把前 ES6 代码转换为 ES6 默认参数值语法的时候要格外小心。

> rest/gather 参数（参见 2.2 节）不能有默认值。所以，尽管 `function foo(...vals=[1,2,3]) {` 这样的用法可能看起来很诱人，但是它并非合法的语法。如果需要的话还是得继续手动提供这种逻辑。

默认值表达式

函数默认值可以不只是像 `31` 这样的简单值；它们可以是任意合法表达式，甚至是函数调用。

```
function bar(val) {
    console.log( "bar called!" );
    return y + val;
}

function foo(x = y + 3, z = bar( x )) {
    console.log( x, z );
}

var y = 5;
foo();                  // "bar called"
                        // 8 13
foo( 10 );              // "bar called"
                        // 10 15
y = 6;
foo( undefined, 10 );   // 9 10
```

可以看到，默认值表达式是惰性求值的，这意味着它们只在需要的时候运行——也就是说，是在参数的值省略或者为 `undefined` 的时候。

这里有一个微妙的细节，注意函数声明中形式参数是在它们自己的作用域中（可以把它看作是就在函数声明包裹的 `( .. )` 的作用域中），而不是在函数体作用域中。这意味着在默认值表达式中的标识符引用首先匹配到形式参数作用域，然后才会搜索外层作用域。参见本系列《你不知道的 JavaScript（上卷）》第一部分可以获取更多信息。

考虑：

```
var w = 1, z = 2;

function foo( x = w + 1, y = x + 1, z = z + 1 ) {
    console.log( x, y, z );
}

foo();                  // ReferenceError
```

w + 1 默认值表达式中的 w 在形式参数列表作用域中寻找 w，但是没有找到，所以就使用外层作用域的 w。接下来，x + 1 默认值表达式中的 x 找到了形式参数作用域中的 x，很幸运这里 x 已经初始化了，所以对 y 的赋值可以正常工作。

但是，z + 1 中的 z 发现 z 是一个此刻还没初始化的参数变量，所以它永远不会试图从外层作用域寻找 z。

正如我们在 2.1.1 节中提到的，ES6 引入了 TDZ，它防止变量在未初始化的状态下被访问。因此，z + 1 默认值表达式会抛出一个 TDZReferenceError 错误。

默认值表达式甚至可以是 inline 函数表达式调用——一般称为立即调用函数表达式（IIFE），但这对于代码明晰性没什么好处。

```
function foo( x =
    (function(v){ return v + 11; })( 31 )
) {
    console.log( x );
}

foo();          // 42
```

IIFE（或者其他任何执行 inline 函数表达式）适用于默认值表达式的情况是很少见的。如果你发现自己需要这么做，那么一定要后退一步重新思考一下！

如果这个 IIFE 试图访问标识符 x，并且没有声明自己的 x 的话，也会引发 TDZ 错误，就像前面讨论的一样。

前面代码中的默认值表达式是一个 IIFE，因为这是一个通过 (31) 立即在线执行的函数。如果没有这一部分，那么赋给 x 的默认值就会是一个函数引用本身，可能就像默认回调那样。这个模式在某些情况下可能是很有用的，比如：

```
function ajax(url, cb = function(){}) {
    // ..
}

ajax( "http://some.url.1" );
```

在这个例子中，我们需要在没有指定其他函数情况下的默认 cb 是一个没有操作的空函数调用。这个函数表达式就是一个函数引用，而不是函数调用本身（后面没有调用形式 ()），这实现了我们的目标。

从 JavaScript 的早期开始，就有一个不为人知但是很有用的技巧可以使用：Function.prototype 本身就是一个没有操作的空函数。所以，这个声明可以是 cb = Function.prototype，这样就省去了在线函数表达式的创建过程。

2.4 解构

ES6 引入了一个新的语法特性，名为**解构**（destructuring），把这个功能看作是一个**结构化赋值**（structured assignment）方法，可能会容易理解一些。考虑：

```
function foo() {
    return [1,2,3];
}

var tmp = foo(),
    a = tmp[0], b = tmp[1], c = tmp[2];

console.log( a, b, c );             // 1 2 3
```

可以看到，我们构造了一个手动赋值，把 `foo()` 返回数组中的值赋给独立变量 `a`、`b` 和 `c`，为了实现这一点，我们（不幸地）需要一个临时变量 `tmp`。

类似地，对于对象可以像下面这么做：

```
function bar() {
    return {
        x: 4,
        y: 5,
        z: 6
    };
}

var tmp = bar(),
    x = tmp.x, y = tmp.y, z = tmp.z;

console.log( x, y, z );            // 4 5 6
```

`tmp.x` 属性值赋给了变量 `x`，同样地，`tmp.y` 赋给了 `y`，`tmp.z` 赋给了 `z`。

可以把将数组或者对象属性中带索引的值手动赋值看作结构化赋值。ES6 为解构新增了一个专门语法，专用于**数组解构**和**对象解构**。这个语法消除了前面代码中对临时变量 `tmp` 的需求，使代码简洁很多。考虑：

```
var [ a, b, c ] = foo();
var { x: x, y: y, z: z } = bar();

console.log( a, b, c );             // 1 2 3
console.log( x, y, z );             // 4 5 6
```

你可能更习惯 `[a,b,c]` 这样的语法作为要赋的值出现在赋值操作符 `=` 的右侧。

解构语法对称地翻转了这个模式，于是赋值符 `=` 左侧的 `[a,b,c]` 被当作某种“模式”，用来把右侧数组值解构赋值到独立的变量中。

类似地，{ x: x, y: y, z: z } 指定了一个把 bar() 的对象值分解为独立的变量赋值的“模式”。

2.4.1 对象属性赋值模式

让我们继续深入探讨一下前面代码中的语法 { x: x, .. }。实际上，如果属性名和要赋值的变量名相同，这种语法还可以更简短一些：

```
var { x, y, z } = bar();

console.log( x, y, z );            // 4 5 6
```

很酷吧，是不是？

但是 { x, .. } 是省略掉了 x: 部分还是 : x 部分呢？实际上我们使用这个缩写语法的时候是略去了 x: 部分。看起来这似乎是无关紧要的细节，但不久以后你就会发现这一点的重要性。

如果可以使用这种简短形式，谁还会再用那种更冗长的形式呢？但是更长的形式支持把属性赋给非同名变量，实际上有时候这是非常有用的：

```
var { x: bam, y: baz, z: bap } = bar();

console.log( bam, baz, bap );      // 4 5 6
console.log( x, y, z );            // ReferenceError
```

关于对象解构形式的这个变体有一个很微妙、但是极其重要的细节需要理解。为了说明为什么要对这一点格外小心，我们考虑一下下面指定一般对象字面值的“模式”：

```
var X = 10, Y = 20;

var o = { a: X, b: Y };

console.log( o.a, o.b );           // 10 20
```

在 { a: X, b: Y } 中，我们知道 a 是对象属性，而 X 是要赋给它的值。换句话说，这个语法模式是 target: source，或者更明确地说是 property-alias：value。因为它和赋值符 = 的模式一样都是 target = source，所以我们很直观地理解了这一点。

但是，在使用对象解构赋值的时候——也就是说，把看起来像是对象字面值的语法 { .. } 放在 = 运算符的左侧——反转了 target: source 模式。

回忆一下：

```
var { x: bam, y: baz, z: bap } = bar();
```

这里的语法模式是 souce: target（或者说是 value:variable-alias）。x: bam 表示 x 属性是源值，而 bam 是要赋值的目标变量。换句话说，对象字面值是 target <-- source，而对

象解构赋值是 `source --> target`。看到这里是如何反转了吧？

还可以这么看待这种语法形式，可能会帮助理解，减少迷惑。考虑：

```
var aa = 10, bb = 20;

var o = { x: aa, y: bb };
var     { x: AA, y: BB } = o;

console.log( AA, BB );              // 10 20
```

在 `{ x: aa, y: bb }` 这一行，`x` 和 `y` 表示对象的属性。在 `{ x: AA, y: BB }` 这一行，`x` 和 `y` 也代表对象属性。

还记得前面指出过 `{ x, .. }` 是省略了 `x:` 部分吗？在这两行里，如果去掉代码中的 `x:` 和 `y:` 这两部分，只剩下 `aa, bb` 和 `AA, BB`，其效果就是——只是概念上，而不是实际上——从 `aa` 赋值给 `AA`，从 `bb` 赋值给 `BB`。

所以，对称性可能会帮助解释为什么这个 ES6 特性的语法模式有意进行了反转。

> 我更喜欢解构赋值的语法是 `{ AA: x , BB: y }`，因为这样两种用法都会保留人们更熟悉的 `target: souce` 模式的一致性。唉，我得训练自己的大脑习惯这个反转，部分读者肯定也是这样。

2.4.2 不只是声明

现在我们已经在 `var` 声明中应用了解构赋值（当然，也可以使用 `let` 和 `const`），但是解构是一个通用的赋值操作，不只是声明。

考虑：

```
var a, b, c, x, y, z;

[a,b,c] = foo();
( { x, y, z } = bar() );

console.log( a, b, c );                 // 1 2 3
console.log( x, y, z );                 // 4 5 6
```

这些变量可能是已经声明的，这样的话解构就只用于赋值，就像这里我们看到的。

> 特别对于对象解构形式来说，如果省略了 `var/let/const` 声明符，就必须把整个赋值表达式用 `( )` 括起来。因为如果不这样做，语句左侧的 `{..}` 作为语句中的第一个元素就会被当作是一个块语句而不是一个对象。

实际上，赋值表达式 (a、y 等) 并不必须是变量标识符。任何合法的赋值表达式都可以。举例来说：

```
var o = {};

[o.a, o.b, o.c] = foo();
( { x: o.x, y: o.y, z: o.z } = bar() );

console.log( o.a, o.b, o.c );       // 1 2 3
console.log( o.x, o.y, o.z );       // 4 5 6
```

甚至可以在解构中使用计算出的属性表达式。考虑：

```
var which = "x",
    o = {};

( { [which]: o[which] } = bar() );

console.log( o.x );                 // 4
```

`[which]:` 这一部分是计算出的属性，结果是 `x`——要从涉及的对象解构出来作为赋值源的属性。`o[which]` 部分就是一个普通的对象键值引用，等价于 `o.x` 作为赋值的目标。

可以用一般的赋值来创建对象映射 / 变换，比如：

```
var o1 = { a: 1, b: 2, c: 3 },
    o2 = {};

( { a: o2.x, b: o2.y, c: o2.z } = o1 );

console.log( o2.x, o2.y, o2.z );    // 1 2 3
```

也可以把一个对象映射为一个数组，比如：

```
var o1 = { a: 1, b: 2, c: 3 },
    a2 = [];

( { a: a2[0], b: a2[1], c: a2[2] } = o1 );

console.log( a2 );                  // [1,2,3]
```

或者反过来：

```
var a1 = [ 1, 2, 3 ],
    o2 = {};

[ o2.a, o2.b, o2.c ] = a1;

console.log( o2.a, o2.b, o2.c );    // 1 2 3
```

还可以把一个数组重排序到另一个：

```
var a1 = [ 1, 2, 3 ],
    a2 = [];

[ a2[2], a2[0], a2[1] ] = a1;

console.log( a2 );                  // [2,3,1]
```

甚至可以不用临时变量解决“交换两个变量”这个经典问题：

```
var x = 10, y = 20;

[ y, x ] = [ x, y ];

console.log( x, y );                // 20 10
```

注意：除非需要把所有的赋值表达式都当作声明，否则不应该在赋值中混入声明。不然会出现语法错误。这也是前面我为什么不得不在 `[ a2[0], .. ] = ...` 解构赋值中把 `var a2 = []` 分离出来。语句 `var [ a2[0], .. ] = ..` 是不合法的，因为 `a2[0]` 不是有效的声明标识符；显然它也不会隐式地创建一个 `var a2 = []` 声明。

2.4.3 重复赋值

对象解构形式允许多次列出同一个源属性（持有值类型任意）。举例来说：

```
var { a: X, a: Y } = { a: 1 };

X;  // 1
Y;  // 1
```

这也意味着可以解构子对象 / 数组属性，同时捕获子对象 / 类的值本身。考虑：

```
var { a: { x: X, x: Y }, a } = { a: { x: 1 } };

X;  // 1
Y;  // 1
a;  // { x: 1 }

( { a: X, a: Y, a: [ Z ] } = { a: [ 1 ] } );

X.push( 2 );
Y[0] = 10;

X;  // [10,2]
Y;  // [10,2]
Z;  // 1
```

关于解构还有一点需要注意：可能你会忍不住要在同一行中列出所有的解构赋值，就像目前为止我们给出的所有示例一样。但是，更好的思路是把解构赋值模式分散在多行中，并

使用适当的缩进——就像使用 JSON 或者对象字面值时一样——这是为了可读性。

```
// 令人费解：
var { a: { b: [ c, d ], e: { f } }, g } = obj;

// 更好的版本：
var {
     a: {
        b: [ c, d ],
        e: { f }
    },
    g
} = obj;
```

记住：解构的目的不只是为了打字更少，而是为了可读性更强。

解构赋值表达式

对象或者数组解构的赋值表达式的完成值是所有右侧对象 / 数组的值。考虑：

```
var o = { a:1, b:2, c:3 },
    a, b, c, p;

p = { a, b, c } = o;

console.log( a, b, c );         // 1 2 3
p === o;                        // true
```

在前面的代码中，p 赋值为对象 o 的引用，而不是 a、b 或者 c 的值之一。数组解构也是这样：

```
var o = [1,2,3],
    a, b, c, p;

p = { a, b, c } = o;

console.log( a, b, c );         // 1 2 3
p === o;                        // true
```

通过持有对象 / 数组的值作为完成值，可以把解构赋值表达式组成链：

```
var o = { a:1, b:2, c:3 },
    p = [4,5,6],

    a, b, c, x, y, z;

( {a} = {b,c} = o );
[x,y] = [z] = p;

console.log( a, b, c );         // 1 2 3
console.log( x, y, z );         // 4 5 4
```

2.5 太多，太少，刚刚好

对于数组解构赋值和对象解构赋值来说，你不需要把存在的所有值都用来赋值。举例来说：

```
var [,b] = foo();
var { x, z } = bar();

console.log( b, x, z );            // 2 4 6
```

foo() 返回的值 1 和 3 被丢弃了，bar() 返回的值 5 也是一样。

类似地，如果为比解构 / 分解出来的值更多的值赋值，那么就像期望的一样，多余的值会被赋为 undefined：

```
var [,,c,d] = foo();
var { w, z } = bar();

console.log( c, z );                // 3 6
console.log( d, w );                // undefined undefined
```

这个特性是符合前面介绍的“undefined 就是缺失”原则的。

本章前面我们介绍了 ... 运算符，了解到有时候它可以用于把数组中的值散开为独立的值，有时候也可以用于做相反的动作：把一组值组合到一起成为一个数组。

除了在函数声明中的 gather/rest 用法，... 也可以执行解构赋值同样的动作。要展示这一点，我们先来回忆一下本章前面的这一段代码：

```
var a = [2,3,4];
var b = [ 1, ...a, 5 ];

console.log( b );                   // [1,2,3,4,5]
```

这里我们看到 ...a 把 a 展开，因为它出现在 [..] 数组值的位置。如果 ...a 出现在数组解构的位置，就执行集合操作：

```
var a = [2,3,4];
var [ b, ...c ] = a;

console.log( b, c );                // 2 [3,4]
```

var [..] = a 解构赋值展开 a 用来给 [..] 中指定的模式赋值。第一部分名为 b 得到 a 中的第一个值（2）。然后则是 ...c 收集了其余的值（3 和 4）赋给一个名为 c 的数组。

> 我们已经看到了 ... 是如何和数组一起工作的，但是如果是和对象呢？这并非 ES6 的特性，但可以参考第 8 章“ES6 之后”，其中讨论了一个可能的新特性，即如何通过 ... 展开和集合对象。

2.5.1　默认值赋值

使用与前面默认函数参数值类似的 = 语法，解构的两种形式都可以提供一个用来赋值的默认值。

考虑：

```
var [ a = 3, b = 6, c = 9, d = 12 ] = foo();
var { x = 5, y = 10, z = 15, w = 20 } = bar();

console.log( a, b, c, d );          // 1 2 3 12
console.log( x, y, z, w );          // 4 5 6 20
```

可以组合使用默认值赋值和前面介绍的赋值表达式语法。举例来说：

```
var { x, y, z, w: WW = 20 } = bar();

console.log( x, y, z, WW );         // 4 5 6 20
```

如果在解构中使用一个对象或者数组作为默认值的话，注意不要绕晕了自己（或者其他阅读你代码的开发者）。你有可能会写出非常晦涩难懂的代码：

```
var x = 200, y = 300, z = 100;
var o1 = { x: { y: 42 }, z: { y: z } };

( { y: x = { y: y } } = o1 );
( { z: y = { y: z } } = o1 );
( { x: z = { y: x } } = o1 );
```

能从上面的代码中看出 x、y 和 z 最后的值是什么吗？我觉得你可能需要花点时间认真思考一下。这里给出了最终答案。

```
console.log( x.y, y.y, z.y );       // 300 100 42
```

记住这一点：解构很不错也可以很有用，但它也是一把利剑，如果不明智使用的话可能会伤了自己（的大脑）。

2.5.2　嵌套解构

如果解构的值中有嵌套的对象或者数组，也可以解构这些嵌套的值：

```
var a1 = [ 1, [2, 3, 4], 5 ];
var o1 = { x: { y: { z: 6 } } };

var [ a, [ b, c, d ], e ] = a1;
var { x: { y: { z: w } } } = o1;

console.log( a, b, c, d, e );        // 1 2 3 4 5
console.log( w );                    // 6
```

可以把嵌套解构当作一种展平对象名字空间的简单方法。举例来说：

```
var App = {
    model: {
        User: function(){ .. }
    }
};

// 不用：
// var User = App.model.User;

var { model: { User } } = App;
```

2.5.3 解构参数

你能在下面的代码中找到其中的赋值吗？

```
function foo(x) {
    console.log( x );
}

foo( 42 );
```

这里的赋值某种程度上说是隐藏的：在执行 foo(42) 的时候把 42（实参）赋给了 x（形参）。如果实参 / 形参配对是一个赋值，那么也就是说它是可以解构的，对吗？当然！

考虑下面的参数数组解构：

```
function foo( [ x, y ] ) {
    console.log( x, y );
}

foo( [ 1, 2 ] );                // 1 2
foo( [ 1 ] );                   // 1 undefined
foo( [] );                      // undefined undefined
```

参数的对象解构也是可以的：

```
function foo( { x, y } ) {
    console.log( x, y );
}

foo( { y: 1, x: 2 } );          // 2 1
foo( { y: 42 } );               // undefined 42
foo( {} );                      // undefined undefined
```

这个技术已经接近于命名参数了（一个 JavaScript 渴望已久的特性！），因为这里对象的属性映射到了同名的解构参数。这也意味着我们免费得到了（任意位置上的）可选参数功能；可以看到，省略“参数”x 就像我们期望的那样工作。

当然，前面介绍的解构的所有变体都可用于参数解构，包括嵌套解构、默认值，等等。解

构还可以和其他的 ES6 函数参数功能同时使用，比如默认参数值和 rest/gather 参数。

下面是一些细节展示（当然不足以囊括所有可能的变体）：

```
function f1([ x=2, y=3, z ]) { .. }
function f2([ x, y, ...z], w) { .. }
function f3([ x, y, ...z], ...w) { .. }

function f4({ x: X, y }) { .. }
function f5({ x: X = 10, y = 20 }) { .. }
function f6({ x = 10 } = {}, { y } = { y: 10 }) { .. }
```

让我们从前面的代码中找一个例子来解释一下：

```
function f3([ x, y, ...z], ...w) {
    console.log( x, y, z, w );
}

f3( [] );                           // undefined undefined [] []
f3( [1,2,3,4], 5, 6 );              // 1 2 [3,4] [5,6]
```

这里使用了两个 ... 运算符，它们都用于收集数组（z 和 w）中的值，当然 ...z 是从第一个数组参数中剩下的值中收集，而 ...w 是从主参数去除第一个值后剩下的值中收集。

1. 解构默认值＋参数默认值

有一点比较微妙需要指出，也是你应该特别注意的，那就是解构默认值和函数参数默认值之间的差别。举例来说：

```
function f6({ x = 10 } = {}, { y } = { y: 10 }) {
    console.log( x, y );
}

f6();                                // 10 10
```

第一眼看上去，我们似乎为参数 x 和 y 都声明了一个默认值 10，虽然是以两种不同的形式。但是，这两种方法在某些情况下的行为是有所不同的，其区别非常微妙。

考虑：

```
f6( {}, {} );                        // 10 undefined
```

稍等，为什么会这样？显然，参数 x 不是作为第一个参数对象的同名属性传入得到默认值 10。

但是为什么 y 值为 undefined？作为函数参数默认值的 { y: 10 } 值是一个对象，而不是解构默认值。因此，它只在第二参数没有传入，或者传入 undefined 的时候才会生效。

在前面的代码中，我们传入了第二个参数（{}），所以没有使用默认值 { y: 10 }，而是在

传入的空对象值 `{}` 上进行 `{ y }` 解构。

现在，比较一下 `{ y } = { y: 10 }` 和 `{ x = 10 } = {}`。

对于 x 这种形式的用法来说，如果第一个函数参数省略或者是 `undefined`，就会应用 `{}` 空对象默认值。然后，在第一个参数位置传入的任何值——或者是默认 `{}` 或者是你传入的任何值——都使用 `{ x = 10 }` 来解构，这会检查是否有 x 属性，如果没有（或者 `undefined`），就会为名为 x 的参数应用默认值 `10`。

缓口气。回头把上面几段重读一遍。我们通过代码来复习一下：

```
function f6({ x = 10 } = {}, { y } = { y: 10 }) {
    console.log( x, y );
}

f6();                               // 10 10
f6( undefined, undefined );         // 10 10
f6( {}, undefined );                // 10 10

f6( {}, {} );                       // 10 undefined
f6( undefined, {} );                // 10 undefined

f6( { x: 2 }, { y: 3 } );           // 2 3
```

看起来 x 参数的默认值特性可能比 y 的情况更符合期望，也更合理一些。因此，理解为什么 `{ x = 10 } = {}` 形式与 `{ y } = { y: 10}` 形式有所区别以及如何进行区别是很重要的。

如果还是有点模糊的话，那么回头再读一遍，然后自己试验一下。花些时间把这个微妙的知识点搞清楚，将来你会感谢自己现在所做的一切的。

2. 嵌套默认：解构并重组

尽管一开始看上去可能很难掌握，这里出现了一个很有趣的为嵌套对象属性设置默认值的技巧：使用对象解构以及我称之为**重组**（restructuring）的技术。

考虑在一个嵌套对象结构内的一组默认值，就像下面这样：

```
// 来自于:
// http://es-discourse.com/t/partial-default-arguments/120/7

var defaults = {
    options: {
        remove: true,
        enable: false,
        instance: {}
    },
    log: {
        warn: true,
        error: true
    }
};
```

假设你有一个名为 config 的对象，已经有了一部分值，但可能不是全部，现在你想要把所有空槽的位置用默认值设定，但又不想覆盖已经存在的部分：

```
var config = {
    options: {
        remove: false,
        instance: null
    }
};
```

当然你可以像过去那样手动实现：

```
config.options = config.options || {};
config.options.remove = (config.options.remove !== undefined) ?
    config.options.remove : defaults.options.remove;
config.options.enable = (config.options.enable !== undefined) ?
    config.options.enable : defaults.options.enable;
...
```

可恶。

还有人会喜欢通过覆盖赋值方法来实现这个任务。你可能会被 ES6 的 Object.assign(..) 工具诱惑（参见第 6 章）先从 defaults 克隆属性，然后用从 config 克隆的属性来覆盖。就像下面这样：

```
config = Object.assign( {}, defaults, config );
```

这看起来好多了，对吧？但是存在一个严重问题！Object.assign(..) 是浅操作，也就是说在复制 defaults.options 的时候，只会复制对象引用，而不会深层复制这个对象的属性到 config.options 对象。需要在对象树的所有层次（某种“递归”）上应用 object.assign(..) 才能得到期望的深层克隆。

> 很多 JavaScript 工具库 / 框架提供了自己的对象深复制支持，但这些方法和它们的使用技巧超出了本部分的讨论范围。

所以让我们来探讨一下带默认值的 ES6 对象解构是否能够帮助实现这一点：

```
config.options = config.options || {};
config.log = config.log || {};
{
    options: {
        remove: config.options.remove = default.options.remove,
        enable: config.options.enable = default.options.enable,
        instance: config.options.instance =
                      default.options.instance
    } = {},
    log: {
```

```
        warn: config.log.warn = default.log.warn,
        error: config.log.error = default.log.error
    } = {}
} = config;
```

我认为这虽然没有提供了虚假保证（实际上只是浅复制）的 `Object.assign(..)` 方法那么优美，但还是要比手动方法要好一点。虽然不幸的是，它还是有点繁复。

前面代码的方法之所以有效，是因为我 hack 了解构和默认值机制，实现了属性 `=== undefined` 检查和赋值决策。这里的巧妙之处在于，我们解构了 `config`（参见代码结尾处的 `= config`），但通过 `config.options.enable` 赋值引用，马上又把所有解构值赋值回到了 `config`。

还是有点繁杂。看我们能不能让实现更简洁一些。

如果确定要解构的所有各种属性都没有重名的话，下面的技巧是最优的。即使不是这样，也可以使用这一技巧，但是就没那么优美了——需要分阶段解构，或者创建唯一局部变量作为临时别名。

如果我们把所有属性彻底解构到顶层变量中，接着就可以立即重组它们来重新构造原来的嵌套对象结构了。

但是，所有这些悬置的临时变量会污染作用域。所以，我们用一个 `{ }` 把这块包起来成为一个块作用域（参见 2.1 节）：

```
// 把defaults合并进config
{
    // (带默认值赋值的)解构
    let {
        options: {
            remove = defaults.options.remove,
            enable = defaults.options.enable,
            instance = defaults.options.instance
        } = {},
        log: {
           warn = defaults.log.warn,
           error = defaults.log.error
        } = {}
    } = config;

    // 重组
    config = {
        options: { remove, enable, instance },
        log: { warn, error }
    };
}
```

这看起来好多了，是不是？

还可以用箭头 IIFE 代替一般的 { } 块和 `let` 声明来实现块封装。解构赋值 / 默认值会被放在参数列表中，而重组的过程会被放在函数体的 `return` 语句中。

重组部分中的 `{ warn, error }` 这种语法形式对你来说可能有点陌生；这称为“简洁属性”，我们将在下一小节介绍！

2.6 对象字面量扩展

ES6 为普通 `{ .. }` 对象字面量新增了几个重要的便利扩展。

2.6.1 简洁属性

你对下面这种形式的对象字面量声明肯定已经非常熟悉了：

```
var x = 2, y = 3,
    o = {
        x: x,
        y: y
    };
```

如果觉得总是要写 `x: x` 令人厌烦的话，那么这里有一个好消息。就是如果你需要定义一个与某个词法标识符同名的属性的话，可以把 `x: x` 简写为 `x`。考虑：

```
var x = 2, y = 3,
    o = {
        x,
        y
    };
```

2.6.2 简洁方法

与刚刚介绍的简洁属性思路类似，为了方便表达，关联到对象字面量属性上的函数也有简洁形式。

老方法：

```
var o = {
    x: function(){
        // ..
    },
    y: function(){
        // ..
    }
}
```

在 ES6 中则可以：

```
var o = {
    x() {
        // ..
    },
    y() {
        // ..
    }
}
```

尽管看上去 `x() { .. }` 就是 `x: function(){ .. }` 的简写形式，但简洁方法有特殊的性质，这是它们的前辈所不具备的；具体来说，就是支持 `super`（参见 2.6.5 节）。

生成器（参见第 4 章）也有一个简洁方法形式：

```
var o = {
    *foo() { .. }
};
```

1. 简洁未命名

这种方便的简写形式是很诱人的，但有一个微妙的细节需要注意。我们分析下面这段前 ES6 代码来展示这一点，这段代码可能令人想要通过简洁方法重构：

```
function runSomething(o) {
    var x = Math.random(),
        y = Math.random();

    return o.something( x, y );
}

runSomething( {
    something: function something(x,y) {
        if (x > y) {
            // 交换x和y的递归调用
            return something( y, x );
        }

        return y - x;
     }
} );
```

这段简单直接的代码就是产生两个随机数字，然后用大的减去小的。但这里重点不是这段代码做了些什么，而是它是如何做的。我们重点关注对象字面量和函数定义，可以看到如下所示：

```
runSomething( {
    something: function something(x,y) {
```

```
        // ..
    }
} );
```

为什么这里既有 something: 又有 function something？这不是重复吗？实际上并不是，二者各有不同的作用，都是必要的。属性 something 使得我们能够通过 o.something(..) 来调用，像是它的公开名称。而第二个 something 是一个词法名称，用于在其自身内部引用这个函数，目的是用于递归。

你能看出为什么 return something(y,x) 这一行需要名称 something 来引用这个函数吗？这个对象没有词法名称，因此它无法使用 return o.something(y,x) 或者某种类似的形式。

当对象字面量有一个标识名称时，这实际上是一个很常见的方法。比如：

```
var controller = {
    makeRequest: function(..){
        // ..
        controller.makeRequest(..);
    }
};
```

这是一个好方法吗？可能是，也可能不是。这里是在假定名称 controller 将会一直指向所需的对象。但是可能实际并非如此——makeRequest(..) 函数并不控制外部代码，因此无法强制这一点。这可能反过来会伤到你自己。

还有一些人可能喜欢采用 this 这种方法来定义：

```
var controller = {
    makeRequest: function(..){
        // ..
        this.makeRequest(..);
    }
};
```

这看起来不错，如果总是通过 controller.makeRequest(..) 调用方法也可以工作。但是，如果要像下面这么做的话，现在就有了一个 this 绑定陷阱：

```
btn.addEventListener( "click", controller.makeRequest, false );
```

当然，可以通过传递 controller.makeRequest.bind(controller) 作为处理函数引用来绑定这个事件。但是这种方法就不怎么吸引人了。

或者，如果内层的 this.makeRequest(..) 调用需要从嵌套函数内部调用会怎样呢？那就得有另外一个 this 绑定，这种情况通常用 var self = this 这种 hack 的方法来解决，就像：

```
var controller = {
    makeRequest: function(..){
```

```
       var self = this;

          btn.addEventListener( "click", function(){
              // ..
              self.makeRequest(..);
          }, false );
      }
};
```

这就更恶心了。

> 关于 this 绑定规则和陷阱的更多信息，参见本系列《你不知道的 JavaScript（上卷）》第二部分的 1~2 章。

好，那么所有这些又和简洁方法有什么关系呢？回想一下我们的 something(..) 方法定义：

```
runSomething( {
    something: function something(x,y) {
        // ..
    }
} );
```

这里的第二个 something 提供了一个超级方便的词法标识符，总是指向这个函数本身，为我们提供了一个完美的用于递归、事件绑定 / 解绑定等的引用——不会和 this 纠缠也不需要并不可靠的对象引用。

非常棒！

所以，现在我们可以把这个函数引用重构为下面的 ES6 简洁方法形式：

```
runSomething( {
    something(x,y) {
        if (x > y) {
            return something( y, x );
        }

        return y - x;
    }
} );
```

第一眼看上去很好，但是这段代码会崩溃。return something(..) 调用将找不到 something 标识符，因此会得到一个 ReferenceError。这是为什么呢？

前面的 ES6 代码片段会被解释为：

```
runSomething( {
    something: function(x,y){
        if (x > y) {
```

```
            return something( y, x );
        }

        return y - x;
    }
} );
```

仔细观察。看到问题所在了吗？这个简洁方法定义意味着 `something: function(x,y)`。看出我们依赖的第二个 `something` 是如何被省略了吗？换句话说，简洁方法意味着匿名函数表达式。

啊，有点恶心。

> 可能你会把 => 箭头函数当作是对这种情况的一个好的解决方案，但是它同样也是不够的，因为它们也是匿名函数表达式。我们将在 2.8 节介绍这一部分。

部分挽回局面的消息是我们的 `something(x,y)` 简洁方法不会是完全匿名的。参见 7.1 节来获取关于 ES6 函数名推导规则的信息。对于我们的递归来说这没有什么帮助，但是至少有助于调试。

所以对于简洁方法能得出什么结论呢？它们简洁方便。但是应该只在不需要它们执行递归或者事件绑定 / 解绑定的时候使用。否则的话，就按照老式的 `something: function something(..)` 方法来定义吧。

大量方法可能会从简洁方法定义中获益，所以这是好消息！只是需要注意这几种有命名问题的情况。

2.ES5 Getter/Setter

严格说来，ES5 定义了 getter/setter 字面量形式，但是没怎么被使用，主要是因为缺少 transpiler 来处理这个新语法（实际上也是 ES5 新增的唯一主要新语法）。所以尽管这并不是一个新的 ES6 特性，我们还是简单介绍一下这种形式，因为很可能在 ES6 及以后它们会得到更广泛地使用。

考虑：

```
var o = {
    __id: 10,
    get id() { return this.__id++; },
    set id(v) { this.__id = v; }
}

o.id;           // 10
o.id;           // 11
o.id = 20;
```

```
o.id;           // 20

// and:
o.__id;         // 21
o.__id;         // 21--保持不变！
```

这些 getter 和 setter 字面量形式也可以出现在类中，参见第 3 章。



可能不是显而易见，实际上 setter 字面量必须有且只有一个声明参数；省略这个参数或者列出多余的都是语法错误。所需的单个参数可以使用解构和默认值（例如，`set id({ id: v = 0 }) { .. }`），但是 gather/rest... 是不允许的（`set id(...v) { .. }`）。

2.6.3　计算属性名

你可能也经历过下面代码片段中的这种情况，其中的一个或多个属性名来自于某个表达式，因此无法用对象字面量表达。

```
var prefix = "user_";

var o = {
    baz: function(..){ .. }
};

o[ prefix + "foo" ] = function(..){ .. };
o[ prefix + "bar" ] = function(..){ .. };
..
```

ES6 对对象字面定义新增了一个语法，用来支持指定一个要计算的表达式，其结果作为属性名。考虑：

```
var prefix = "user_";

var o = {
    baz: function(..){ .. },
    [ prefix + "foo" ]: function(..){ .. },
    [ prefix + "bar" ]: function(..){ .. }
    ..
};
```

对象字面定义属性名位置的 `[ .. ]` 中可以放置任意合法表达式。

计算属性名最常见的用法可能就是和 `Symbols` 共同使用（我们将在 2.13 节中介绍）。比如：

```
var o = {
    [Symbol.toStringTag]: "really cool thing",
    ..
};
```

Symbol.toStringTag 是一个特殊的内置值，我们用 [..] 语法为其求值，所以我们可以把值 "really cool thing" 赋给这个特殊的属性名。

计算属性名也可以作为简洁方法或者简洁生成器的名称出现：

```
var o = {
    ["f" + "oo"]() { .. }    // 计算出的简洁方法
    *["b" + "ar"]() { .. }  // 计算出的简洁生成器
};
```

2.6.4 设定 [[Prototype]]

这里我们不会详细介绍原型，要想了解更多信息，参见本系列《你不知道的 JavaScript（上卷)》第二部分。

有时候在声明对象字面量的时候设定这个对象的 [[Prototype]] 是有用的。下面的用法在很多 JavaScript 引擎中已经作为非标准扩展有一段时间了，而在 ES6 中这已经标准化了：

```
var o1 = {
    // ..
};

var o2 = {
    __proto__: o1,
    // ..
};
```

o2 通过普通的对象字面量声明，但是它也 [[Prototype]] 连接到了 o1。这里的 __proto__ 属性名也可以是字符串 "__proto__"，但是注意它不能是计算属性名结果（参见前一节）。

退一步讲，对 __proto__ 的使用是有争议的。它是 JavaScript 多年前的属性扩展，最后被 ES6 标准化，但似乎标准化得不情不愿。许多开发者认为不应该使用它。实际上，它是在 ES6 的“附录 B”中出现的，这一部分列出的都是 JavaScript 只因兼容性问题不得不标准化的特性。

> 我勉强支持 __proto__ 作为对象字面量定义的一个键值，但我绝对不支持作为对象属性形式来使用它，比如 o.__proto__。这种形式既是 getter 又是 setter（也是由于兼容性的原因)，但是肯定还有更好的选择。参见本系列《你不知道的 JavaScript（上卷)》第二部分。

要为已经存在的对象设定 [[Prototype]]，可以使用 ES6 工具 Object.setPrototypeOf(..)。考虑：

```
var o1 = {
    // ..
```

```
};

var o2 = {
    // ..
};

Object.setPrototypeOf( o2, o1 );
```

我们会在第 6 章再次讨论 Object。“Object.setPrototypeOf(..) 静态函数”一节详细介绍了 Object.setProto typeOf(..)。还可以参考 6.2.4 节了解把 o2 原型关联到 o1 的另外一种形式。

2.6.5　super 对象

通常把 super 看作只与类相关。但是，鉴于 JavaScript 的原型类而非类对象的本质，super 对于普通（plain）对象的简洁方法也一样有效，特性也基本相同。

考虑：

```
var o1 = {
    foo() {
        console.log( "o1:foo" );
    }
};

var o2 = {
    foo() {
        super.foo();
        console.log( "o2:foo" );
    }
};

Object.setPrototypeOf( o2, o1 );

o2.foo();       // o1:foo
                // o2:foo
```

super 只允许在简洁方法中出现，而不允许在普通函数表达式属性中出现。也只允许以 super.XXX 的形式（用于属性 / 方法访问）出现，而不能以 super() 的形式出现。

o2.foo() 方法中的 super 引用静态锁定到 o2，具体说是锁定到 o2 的 [[Prototype]]。基本上这里的 super 就是 Object.getPrototypeOf(o2)——当然会决议到 o1——这是它如何找到并调用 o1.foo() 的过程。

关于 super 的完整细节，参见 3.4 节。

2.7　模板字面量

在这一小节的最开始，首先我要大声指出 ES6 这个特性的名称非常具有误导性，其根据你对单词**模板**（template）的理解而定。

很多开发者会把模板看作是可复用、可渲染的文字片段，就像绝大多数模板引擎（Mustache、Handlebars 等）提供的功能那样。ES6 对 template 这个词的使用可能暗示着某种类似的东西，就像一种声明可以被重新渲染的在线模板字面量的方法。但是，对于这个特性来说，这种看法并不完整。

所以，在继续之前，我要把它重命名为它应该被称为的：**插入字符串字面量**（或者简称为 interpoliteral）。

你已经非常了解，可以用 " 或 ' 界定符声明字符串，也了解这不是（像某些语言中那样的）**smart 字符串**，其中的内容可以插入表达式被重新解析。

但是，ES6 引入了一个新的字符串字面量，使用 ` 作为界定符。这样的字符串字面值支持嵌入基本的字符串插入表达式，会被自动解析和求值。

下面是老的前 ES6 方式：

```
var name = "Kyle";

var greeting = "Hello " + name + "!";

console.log( greeting );            // "Hello Kyle!"
console.log( typeof greeting );     // "string"
```

下面是新的 ES6 方式：

```
var name = "Kyle";

var greeting = `Hello ${name}!`;

console.log( greeting );           // "Hello Kyle!"
console.log( typeof greeting );    // "string"
```

你可以看到，这里在一组字符外用 `..` 来包围，这会被解释为一个字符串字面量，但是其中任何 `${..}` 形式的表达式都会被立即在线解析求值。这种形式的解析求值形式就是**插入**（比模板要精确一些）。

插入字符串字面量表达式的结果就是普通的字符串，值赋给变量 `greeting`。

typeof greeting == "string" 说明了为什么不要把这些实体当作是特殊的模板值这一点很重要，因为不能把这个字面量未求值的形式赋给某个实体然后复用。`..` 字符串字面量更像是 IIFE，因为它会自动展开求值。一个 `..` 字符串字面量的结果就是一个字符串。

插入字符串字面量的一个优点是它们可以分散在多行：

```
var text =
`Now is the time for all good men
to come to the aid of their
country!`;

console.log( text );
// Now is the time for all good men
// to come to the aid of their
// country!
```

插入字符串字面量中的换行（新行）会在字符串值中被保留。

在字面量值中，除非作为明确的转义序列出现，`\r` 回车符（码点 `U+000D`）的值或者回车换行符 `\r\n`（码点 `U+000D` 和 `U+000A`）都会被标准化为 `\n` 换行符（码点 `U+000A`）。但是别担心，这种标准化非常少见，很可能只有在复制粘贴文本到 JavaScript 文件的时候才会出现。

2.7.1　插入表达式

在插入字符串字面量的 `${..}` 内可以出现任何合法的表达式，包括函数调用、在线函数表达式调用，甚至其他插入字符串字面量！

考虑：

```
function upper(s) {
    return s.toUpperCase();
}

var who = "reader";

var text =
`A very ${upper( "warm" )} welcome
to all of you ${upper( `${who}s` )}!`;

console.log( text );
// A very WARM welcome
// to all of you READERS!
```

这里，与 `who + "s"` 的形式相比，内层的 `` `${who}s` `` 插入字符串字面量对我们合并变量 `who` 和字符串 `"s"` 来说会方便一点。嵌套插入字符串字面量在一些情况下是有所帮助的，但是如果你发现需要频繁使用这种形式，那么就要警惕了，不然你会发现自己得嵌套好多层。

如果是这样的话，那么很可能你的字符串产生过程需要进行某种抽象。

> 提醒一下，在代码中使用这些新武器的时候要注意可读性。就像使用默认值表达式和解构赋值表达式的时候一样，能够做某事并不意味着就应该这么做。千万不要过分热衷于 ES6 的新技巧，使得你的代码的“聪明”程度超过了你自己或你的同事。

表达式作用域

这里对于用来在表达式中决议变量的作用域有一点简单说明。我在前面提到过插入字符串字面量有点像是一个 IIFE，这么看也能解释它的作用域特性。

考虑：

```
function foo(str) {
    var name = "foo";
    console.log( str );
}

function bar() {
    var name = "bar";
    foo( `Hello from ${name}!` );
}

var name = "global";

bar();                  // "Hello from bar!"
```

`..` 字符串字面值展开的时候，在函数 bar() 内部，它可用的作用域找到 bar() 的变量 name 的值 "bar"。全局 name 和 foo() 的 name 都不影响结果。换句话说，插入字符串字面量在它出现的词法作用域内，没有任何形式的动态作用域。

2.7.2 标签模板字面量

这里再次重新命名这个特性来明确表达其功能：**标签字符串字面量** (tagged string literal)。

老实说，这是 ES6 提供的一个比较酷的技巧。可能看起来有点奇怪，也可能一眼看上去不是那么实用。但一旦花费一定时间去理解这个功能，标签字符串字面量的用处可能会令你大吃一惊。

举例来说：

```
function foo(strings, ...values) {
    console.log( strings );
    console.log( values );
}
```

```
var desc = "awesome";

foo`Everything is ${desc}!`;
// [ "Everything is ", "!"]
// [ "awesome" ]
```

我们花点时间来思考一下前面代码中到底发生了什么。首先跳出的最不和谐的部分是 foo`Everything...`;。这种形式之前没有出现过。这是什么？

本质上说，这是一类不需要（ .. ）的特殊函数调用。**标签**（tag）部分，即 `..` 字符串字面量之前的 foo 这一部分，是一个要调用的函数值。实际上，它可以是任意结果为函数的表达式，甚至可以是一个结果为另一个函数的函数调用，就像下面这样：

```
function bar() {
    return function foo(strings, ...values) {
        console.log( strings );
        console.log( values );
    }
}

var desc = "awesome";

bar()`Everything is ${desc}!`;
// [ "Everything is ", "!"]
// [ "awesome" ]
```

但是传入为了字符串字面量作为标签被调用的 foo(..) 函数的是什么呢？

第一个参数，名为 strings，是一个由所有普通字符串（插入表达式之间的部分）组成的数组。得到的 strings 数组中有两个值："Everything is" 和 "!"。

我们的例子中使用 ...gather/rest 运算符把其余所有参数值收集到名为 values 的数组中（参见 2.2 节），这是为了方便起见，当然也可以把 strings 参数后面的其余部分都作为独立命名的参数。

收集到 values 数组的参数是已经求值的在字符串字面值中插入表达式的结果。所以显然我们的例子中 values 的唯一元素是 "awesome"。

你可以这样看待这两个数组：values 中的值是分隔符，就好像用它们连接在 strings 中的值，然后把所有这些都连接到一起，就得到了一个完成的插入字符串值。

标签字符串字面量就像是一个插入表达式求值之后，在最后的字符串值编译之前的处理步骤，这个步骤为从字面值产生字符串提供了更多的控制。

一般来说，字符串字面量标签函数（前面代码中的 foo(..)）要计算出一个适当的字符串并将其返回，这样就可以像使用非标签字符串字面量一样把标签字符串字面量作为一个值来使用了：

```
function tag(strings, ...values) {
    return strings.reduce( function(s,v,idx){
        return s + (idx > 0 ? values[idx-1] : "") + v;
    }, "" );
}

var desc = "awesome";

var text = tag`Everything is ${desc}!`;

console.log( text );            // Everything is awesome!
```

在这段代码中，tag(..) 是一个直通操作，因为它不执行任何具体修改，而只是使用 reduce(..) 进行循环，把 strings 和 values 连接到一起，就像非标签字符串字面量所做的一样。

那么有哪些实际应用呢？有许多高级应用已经超出了本部分的讨论范围。但是，这里还是给出了一个简单的思路用来把数字格式化为美元表示法（类似于简单的本地化）：

```
function dollabillsyall(strings, ...values) {
    return strings.reduce( function(s,v,idx){
        if (idx > 0) {
            if (typeof values[idx-1] == "number") {
                // 看，这里也使用了插入字符串字面量！
                s += `$${values[idx-1].toFixed( 2 )}`;
            }
            else {
               s += values[idx-1];
            }
        }

        return s + v;
    }, "" );
}

var amt1 = 11.99,
    amt2 = amt1 * 1.08,
    name = "Kyle";

var text = dollabillsyall
`Thanks for your purchase, ${name}! Your
product cost was ${amt1}, which with tax
comes out to ${amt2}.`

console.log( text );
// Thanks for your purchase, Kyle! Your
// product cost was $11.99, which with tax
// comes out to $12.95.
```

如果在 values 中遇到一个 number 值，就在其之前放一个 "$"，然后用 toFixed(2) 把它格式化为两个十进制数字的形式，否则就让这个值直接通过而不做任何修改。

原始（raw）字符串

在前面的代码中，标签函数接收第一个名为 `strings` 的参数，这是一个数组。但是还包括了一些额外的数据：所有字符串的原始未处理版本。可以像下面这样通过 `.raw` 属性访问这些原始字符串值：

```
function showraw(strings, ...values) {
    console.log( strings );
    console.log( strings.raw );
}

showraw`Hello\nWorld`;
// [ "Hello
// World" ]
// [ "Hello\nWorld" ]
```

原始版本的值保留了原始的转义码 `\n` 序列（`\` 和 `n` 是独立的字符），而处理过的版本把它当作是一个单独的换行符。二者都会应用前面提到过的行结束标准化过程。

ES6 提供了一个内建函数可以用作字符串字面量标签：`String.raw(..)`。它就是传出 `strings` 的原始版本：

```
console.log( `Hello\nWorld` );
// Hello
// World

console.log( String.raw`Hello\nWorld` );
// Hello\nWorld

String.raw`Hello\nWorld`.length;
// 12
```

字符串字面量标签的其他应用包括全球化、本地化等的特殊处理。

2.8 箭头函数

本章前面已经介绍了一些函数中 `this` 绑定的复杂性，同时本系列《你不知道的 JavaScript（上卷）》第二部分中对此也有详细介绍。理解使用普通函数基于 `this` 编程带来的令人沮丧的问题是很重要的，因为这是新的 ES6 箭头函数 `=>` 特性引入的主要动因。

让我们先来展示一下与普通函数相比箭头函数是什么样子：

```
function foo(x,y) {
    return x + y;
}

// 对比

var foo = (x,y) => x + y;
```

箭头函数定义包括一个参数列表（零个或多个参数，如果参数个数不是一个的话要用 (..) 包围起来），然后是标识 =>，函数体放在最后。

所以，在前面的代码中，箭头函数就是 (x,y) => x + y 这一部分，然后这个函数引用被赋给变量 foo。

只有在函数体的表达式个数多于 1 个，或者函数体包含非表达式语句的时候才需要用 { .. } 包围。如果只有一个表达式，并且省略了包围的 { .. } 的话，则意味着表达式前面有一个隐含的 return，就像前面代码中展示的那样。

这里列出了几种不同形式的箭头函数：

```
var f1 = () => 12;
var f2 = x => x * 2;
var f3 = (x,y) => {
    var z = x * 2 + y;
    y++;
    x *= 3;
    return (x + y + z) / 2;
};
```

箭头函数**总是**函数表达式；并不存在箭头函数声明。我们还应清楚箭头函数是匿名函数表达式——它们没有用于递归或者事件绑定 / 解绑定的命名引用——但 7.1 节将会讨论 ES6 中用于调试目的的函数名推导规则。

箭头函数支持普通函数参数的所有功能，包括默认值、解构、rest 参数，等等。

箭头函数语法清晰简洁，这使它们表面上看起来对于编写更简练的代码很有吸引力。于是，几乎所有关于 ES6 的文献（除了本系列）似乎都立即且没有异议地接受了箭头函数作为"新函数"。

可以说这里关于箭头函数的讨论中几乎所有的例子都是简短的单句工具，比如作为回调函数传递给各种工具的那些。举例来说：

```
var a = [1,2,3,4,5];

a = a.map( v => v * 2 );

console.log( a );               // [2,4,6,8,10]
```

这些例子中你使用了这样的在线函数表达式，它们也符合在单个语句中执行一个快速计算并返回结果的模式，这时候比起繁复的 function 关键字和语法，箭头函数确实看起来是更有吸引力的轻量替代工具。

大多数人会**赞叹于**这些示例的简洁，我猜你也是这样！

但是这里我要提醒你，使用箭头函数语法替代其他普通的多行函数，特别是那些通常会被自然表达为函数声明的情况，是不合理的。

回忆一下本章前面的 `dollabillsyall(..)` 字符串字面量标签函数，把它替换为使用 `=>` 语法：

```
var dollabillsyall = (strings, ...values) =>
    strings.reduce( (s,v,idx) => {
        if (idx > 0) {
            if (typeof values[idx-1] == "number") {
                // 看，这里也使用了插入字符串字面量！
                s += `$${values[idx-1].toFixed( 2 )}`;
            }
            else {
                s += values[idx-1];
            }
        }

        return s + v;
    }, "" );
```

在这个例子中，我所做的唯一修改就是去掉了 `function`、`return` 和一些 `{ .. }`，然后插入了 `=>` 和 `var`。对于这段代码来说，可读性有了明显的改进吗？

实际上，我认为缺少 `return` 和外层的 `{ .. }` 一定程度上模糊了 `reduce(..)` 调用是 `dollabillsyall(..)` 函数中唯一的语句，以及它的返回值就是调用的返回值这一事实。另外，有经验的人会在代码中搜索单词 `function` 来寻找作用域的边界，现在则需要寻找 `=>` 标识，这在大段代码中肯定更加难以发现。

虽然不是一条严格的规律，但我认为 `=>` 箭头函数转变带来的可读性提升与被转化函数的长度负相关。这个函数越长，`=>` 带来的好处就越小；函数越短，`=>` 带来的好处就越大。

我认为更合理的做法是只在确实需要简短的在线函数表达式的时候才采用 `=>`，而对于那些一般长度的函数则无需改变。

不只是更短的语法，而是 this

对 `=>` 的关注多数都在于从代码中去掉 `function`、`return` 和 `{ .. }` 节省了那些宝贵的键盘输入。

但是，目前为止我们省略了一个重要的细节。这一节开头提到 `=>` 函数与 `this` 绑定行为紧密相关。实际上，`=>` 箭头函数的**主要设计目的**就是以特定的方式改变 `this` 的行为特性，解决 `this` 相关编码的一个特殊而又常见的痛点。

节省的输入字符是一条红鲱鱼（转移注意力的东西），至少是误导性的。

让我们回顾一下本章前面的另外一个例子：

```
var controller = {
    makeRequest: function(..){
        var self = this;

        btn.addEventListener( "click", function(){
            // ..
           self.makeRequest(..);

        }, false );
    }
};
```

我们使用了 `var self = this` 这一 hack，然后引用 `self.makeRequest(..)`，因为在我们传入 `addEventListener(..)` 的回调函数内部，`this` 绑定和 `makeRequest(..)` 本身的 `this` 绑定是不同的。换句话说，因为 `this` 绑定是动态的，我们通过变量 `self` 依赖于词法作用域的可预测性。

这里我们终于可以看到 `=>` 箭头函数的主要设计特性了。在箭头函数内部，`this` 绑定不是动态的，而是词法的。在前面的代码中，如果使用箭头函数作为回调，`this` 则如我们所愿是可预测的。

考虑：

```
var controller = {
    makeRequest: function(..){
        btn.addEventListener( "click", () => {
            // ..
            this.makeRequest(..);
        }, false );
    }
};
```

前面代码的箭头函数回调中的词法 `this` 现在与封装的函数 `makeRequest(..)` 指向同样的值。换句话说，`=>` 是 `var self = this` 的词法替代形式。

在通常需要 `var self = this`（或者换种形式，如函数 `.bind(this)` 调用）的时候，基于运行原则相同，可以把 `=>` 箭头函数作为一个很好的替代。看起来不错，是吗？

但没有这么简单。

假使用 `=>` 替代 `var self = this` 或者 `.bind(this)` 的情况有所帮助，那么猜一下如果在一个支持 `this` 的函数中使用 `=>`，而这个函数**不**需要 `var self = this` 会怎样呢？你可能已经猜到了，这会把事情搞乱。没错。

考虑：

```
var controller = {
    makeRequest: (..) => {
        // ..
        this.helper(..);
    },
    helper: (..) => {
        // ..
    }
};

controller.makeRequest(..);
```

尽管我们以 `controller.makeRequest(..)` 的形式调用，`this.helper` 引用还是会失败，因为这里的 `this` 并不像平常一样指向 `controller`。那么它指向哪里呢？它是从包围的作用域中词法继承而来的 `this`。在前面的代码中也就是全局作用域，其中 `this` 指向那个全局对象。

除了词法 `this`，箭头函数还有词法 `arguments`——它们没有自己的 `arguments` 数组，而是继承自父层——词法 `super` 和 `new.target` 也是一样（参见 3.4 节）。

所以现在我们可以给出一组更详细的 `=>` 适用时机的规则。

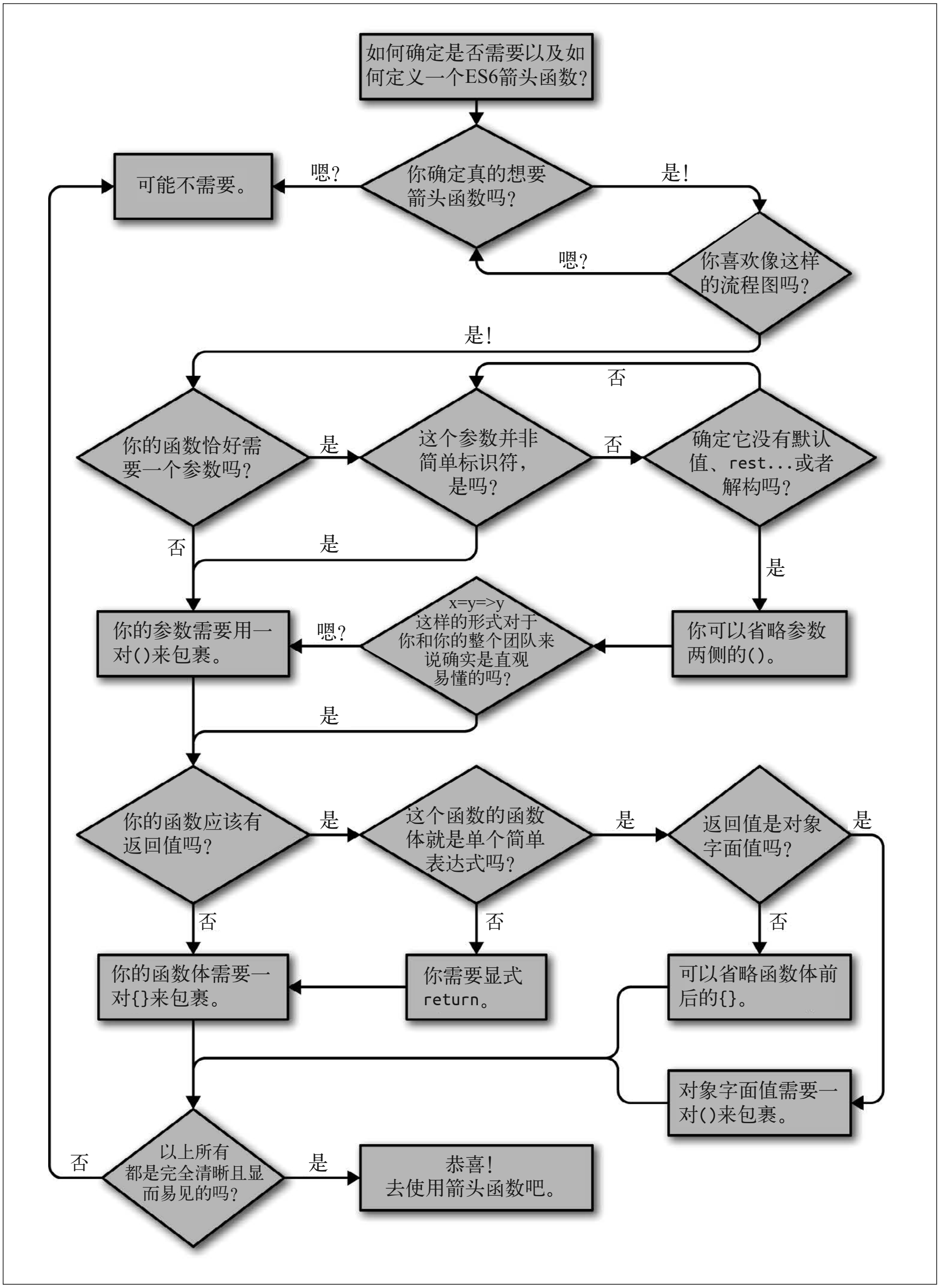
- 如果你有一个简短单句在线函数表达式，其中唯一的语句是 `return` 某个计算出的值，**且**这个函数内部没有 `this` 引用，**且**没有自身引用（递归、事件绑定 / 解绑定），**且**不会要求函数执行这些，那么可以安全地把它重构为 `=>` 箭头函数。
- 如果你有一个内层函数表达式，依赖于在包含它的函数中调用 `var self = this` hack 或者 `.bind(this)` 来确保适当的 `this` 绑定，那么这个内层函数表达式应该可以安全地转换为 `=>` 箭头函数。
- 如果你的内层函数表达式依赖于封装函数中某种像 `var args = Array.prototype.slice.call(arguments)` 来保证 `arguments` 的词法复制，那么这个内层函数应该可以安全地转换为 `=>` 箭头函数。
- 所有的其他情况——函数声明、较长的多语句函数表达式、需要词法名称标识符（递归等）的函数，以及任何不符合以上几点特征的函数——一般都应该避免 `=>` 函数语法。

底线：`=>` 是关于 `this`、`arguments` 和 `super` 的词法绑定。这个 ES6 的特性设计用来修正一些常见的问题，而不是 bug、巧合或者错误。

不要相信那些宣传所说的 `=>` 主要甚至绝大多数关注点在于少打字。不管是少打字还是多打字，你应该精确了解自己输入的每个字符的目的所在。

> 如果你有一个函数，出于明确的原因其不适合 `=>` 箭头函数，但是它被声明为对象字面量的一部分，回忆一下 2.6.2 节的内容，要使函数语法简洁还有其他可选的方法。

这里用一个可视化的决策图来展示如何 / 为什么采用箭头函数：



2.9 for..of 循环

ES6 在把 JavaScript 中我们熟悉的 for 和 for..in 循环组合起来的基础上，又新增了一个 for..of 循环，在**迭代器**产生的一系列值上循环。

for..of 循环的值必须是一个 iterable，或者说它必须是可以转换 / 封箱到一个 iterable 对象的值（参见本系列《你不知道的 JavaScript（中卷）》第一部分）。iterable 就是一个能够产生迭代器供循环使用的对象。

我们来对比一下 for..of 和 for..in 以展示其中的区别：

```
var a = ["a","b","c","d","e"];

for (var idx in a) {
    console.log( idx );
}
// 0 1 2 3 4

for (var val of a) {
    console.log( val );
}
// "a" "b" "c" "d" "e"
```

可以看到，for..in 在数组 a 的键 / 索引上循环，而 for..of 在 a 的值上循环。

下面是前面代码中的前 ES6 版本的 for..of 形式：

```
var a = ["a","b","c","d","e"],
    k = Object.keys( a );

for (var val, i = 0; i < k.length; i++) {
    val = a[ k[i] ];
    console.log( val );
}
// "a" "b" "c" "d" "e"
```

这里是 ES6 的但是不用 for..of 的等价代码，也可以用来展示如何手动在迭代器上迭代（参见 3.1 节）：

```
var a = ["a","b","c","d","e"];

for (var val, ret, it = a[Symbol.iterator]();
    (ret = it.next()) && !ret.done;
) {
    val = ret.value;
    console.log( val );
}
// "a" "b" "c" "d" "e"
```

在底层，for..of 循环向 iterable 请求一个迭代器（通过内建的 Symbol.iterator，参见 7.3 节），然后反复调用这个迭代器把它产生的值赋给循环迭代变量。

JavaScript 中默认为（或提供）iterable 的标准内建值包括：

- Arrays
- Strings
- Generators（参见第 3 章）
- Collections / TypedArrays (参见第 5 章）

> 默认情况下平凡对象并不适用 `for..of` 循环。因为它们并没有默认的迭代器，这是有意设计的特性，而不是错误。但这里我们不会深入推导这一结论。在 3.1 节中，我们将会介绍如何为自己的对象定义迭代器，这样就可以使 `for..of` 在任何对象上循环，得到一组我们定义的值。

下面是在原生字符串的字符上迭代的方式：

```
for (var c of "hello") {
    console.log( c );
}
// "h" "e" "l" "l" "o"
```

原生字符串值 `"hello"` 被强制类型转换 / 封箱到等价的 `String` 封装对象中，而这默认是一个 iterable。

在 `for (XYZ of ABC)..` 中，和 `for` 以及 `for..in` 循环中的语句一样，`XYZ` 语句可以是赋值表达式也可以是声明。所以可以这么做：

```
var o = {};

for (o.a of [1,2,3]) {
    console.log( o.a );
}
// 1 2 3

for ({x: o.a} of [ {x: 1}, {x: 2}, {x: 3} ]) {
  console.log( o.a );

  }
// 1 2 3
```

和其他循环一样，`for..of` 循环也可以通过 `break`、`continue`、`return`（如果在函数中的话）提前终止，并抛出异常。在所有这些情况中，如果需要的话，都会自动调用迭代器的 `return(..)` 函数（如果存在的话）让迭代器执行清理工作。

> 关于 iterable 和迭代器的完成信息参见 3.1 节。

2.10 正则表达式

让我们面对这样一个事实：JavaScript 中的正则表达式很长时间以来基本没有什么变化。所以当 ES6 中终于增加了一些新技巧，真是太好了。这里会概括介绍一下新增的特性，但是全面的正则表达式主题是很复杂的，如果想要学习的话需要阅读专门介绍这个主题的章节 / 书籍（这样的书籍有很多！）。

2.10.1 Unicode 标识

我们会在 2.12 节详细介绍这一主题。这里只是简单介绍 ES6+ 正则表达式新的 `u` 标识，这个标识会为表达式打开 Unicode 匹配。

JavaScript 字符串通常被解释成 16 位字符序列，这些字符对应**基本多语言平面**（Basic Multilingual Plane，BMP）（http://en.wikipedia.org/wiki/Plane_%28Unicode %29）中的字符。但还有很多 UTF-16 字符在这个范围之外，所以字符串中还可能包含这些多字节字符。

在 ES6 之前，正则表达式只能基于 PMB 字符匹配，这意味着那些扩展字符会被当作两个独立的字符来匹配。通常这不是理想的做法。

所以，在 ES6 中，`u` 标识符表示正则表达式用 Unicode（UTF-16）字符来解释处理字符串，把这样的扩展字符当作单个实体来匹配。

> 并不像其名字所暗示的，“UTF-16”不完全是 16 位的。现代的 Unicode 使用 21 位来表示，像 UTF-8、UTF-16 这样的标准只是大概表明了用多少位来表示一个字符。

举一个例子（直接来自于 ES6 规范）：音乐符号 𝄞（高音符号）是 Unicode 码点为 U+1D11E (0x1D11E)。

如果这个符号出现在正则表达式中（比如 `/𝄞/`），标准 PMB 解释在匹配的时候会将其解释为两个独立的字符（0xD834 和 0xDD1E）。而新的 ES6 支持 Unicode 模式，意味着 `/𝄞/u`（或者转义符 Unicode 形式 `/\u{1D11E}/u`）会把字符串中的 `"𝄞"` 作为一个单独的匹配字符。

你可能会问这又有什么关系呢？在非 Unicode 的 BMP 模式，这个模式会被当作两个独立的字符，但是仍然会匹配到包含有 `"𝄞"` 的字符串，像下面这样试一下就会了解：

```
/𝄞/.test( "𝄞-clef" );      // true
```

影响的是匹配部分的长度。举例来说：

```
/^.-clef/ .test( "𝄞-clef" );    // false
/^.-clef/u.test( "𝄞-clef" );    // true
```

模式中的 `^.-clef` 表明只匹配起始处并在普通文本 `"-clef"` 之前有一个字符的情形。在标准的 BMP 模式中，匹配会失败（两个字符），如果用 `u` 标识打开 Unicode 模式，匹配则会成功（单个字符）。

还有一点需要注意，`u` 标识使得 `+` 和 `*` 这样的量词把整个 Unicode 码点作为单个字符而应用，而不仅仅是应用于字符的**低位**（也就是符号的最右部分）。在字符类内部出现的 Unicode 字符也是一样，比如 `/[💩-💫]/u`。

> 正则表达式中的 `u` 标识符行为特性还有很多重要的技术细节，关于这些 Mathias Bynens（https://twitter.com/mathias）在这篇文章（https://mathiasbynens.be/notes/es6-unicode-regex）中有详细的介绍。

2.10.2　定点标识

ES6 正则表达式中另外一个新增的标签模式是 `y`，通常称为“定点（sticky）模式”。**定点**主要是指在正则表达式的起点有一个虚拟的锚点，只从正则表达式的 `lastIndex` 属性指定的位置开始匹配。

为了说明这一点，我们来考虑两个正则表达式——第一个没有定点模式，而第二个有：

```
var re1 = /foo/,
    str = "++foo++";

re1.lastIndex;          // 0
re1.test( str );        // true
re1.lastIndex;          // 0--没有更新

re1.lastIndex = 4;
re1.test( str );        // true--被忽略的lastIndex
re1.lastIndex;          // 4--没有更新
```

从上面的代码中可以观察到 3 点。

- `test(..)` 并不关心 `lastIndex` 的值，总是从输入字符串的起始处开始执行匹配。
- 因为我们的模式并没有起始锚点 `^`，对 `"foo"` 的搜索从整个字符串向前寻找匹配。
- `test(..)` 不更新 `lastIndex`。

现在，我们试验一下定点模式正则表达式：

```
var re2 = /foo/y,     // <-- 注意定点标识y
    str = "++foo++";

re2.lastIndex;        // 0
re2.test( str );      // false--0处没有找到"foo"
re2.lastIndex;        // 0
```

```
re2.lastIndex = 2;
re2.test( str );     // true
re2.lastIndex;       // 5--更新到前次匹配之后位置

re2.test( str );     // false
re2.lastIndex;       // 0--前次匹配失败后重置
```

下面是关于定点模式的新的观察结果。

- `test(..)` 使用 `lastIndex` 作为 `str` 中精确而且唯一的位置寻找匹配。不会向前移动去寻找匹配——要么匹配位于 `lastIndex` 位置上，要么就没有匹配。
- 如果匹配成功，`test(..)` 会更新 `lastIndex` 指向紧跟匹配内容之后的那个字符。如果匹配失败，`test(..)` 会把 `lastIndex` 重置回 `0`。

一般的没有用 `^` 限制输入起始点匹配的非定点模式可以自由地在输入字符串中向前移动寻找匹配内容。而定点模式则限制了模式只能从 `lastIndex` 开始匹配。

正如这一小节开始所提到的，另一种理解思路是把 `y` 看作一个在模式开始处的虚拟锚点，限制模式（也就是限制匹配的起点）相对于 `lastIndex` 的位置。

> 关于这个主题的已有文献中，还有一种思路是 `y` 意味着一个模式中的 `^`（起点）锚点。这并不精确，后面我们在“定点锚点”一节中会详细解释。

1. 定点定位

因为 `y` 模式不能向前移动搜索匹配，所以要把 `y` 用于重复匹配，就不得不手动保证 `lastIndex` 指向正确的位置，这种局限性有点奇怪。

这里有一个可能的应用场景：如果确定你关心的匹配总是在某个数字的整数倍位置上（例如：`0`、`10`、`20` 等），就可以构造一个受限的模式来匹配所关心的内容，然后每次在匹配之前手动把 `lastIndex` 设定到这些固定的位置。

考虑：

```
var re = /f../y,
    str = "foo       far       fad";

str.match( re );        // ["foo"]

re.lastIndex = 10;
str.match( re );        // ["far"]

re.lastIndex = 20;
str.match( re );        // ["fad"]
```

但是，如果要处理的字符串并没有这样格式化在固定的位置上，每次匹配之前都要计算出

需要把 lastIndex 设置成什么值可就不那么合理了。

这里有一个技术细节可以考虑用于解决这个问题。y 要求 lastIndex 精确位于每个匹配发生的位置。但是并没有严格要求手动设置这个 lastIndex。

相反，可以以自己的方式构造一个表达式，让它在主匹配之前能够捕获从你关注的内容之后，恰好直到下一次关注内容之前。

因为 lastIndex 会设定为匹配结尾处的下一个字符，所以如果匹配了直到这个点的所有东西，lastIndex 就总会位于 y 模式下次要开始的正确位置。



如果无法预测输入字符串的结构的模式，那么这个技术就不合适，可能无法应用 y。

可以应用 y 模式在字符串中执行重复匹配最适合的场景可能就是结构化的输入字符串。考虑：

```
var re = /\d+\.\s(.*?)(?:\s|$)/y
    str = "1. foo 2. bar 3. baz";

str.match( re );        // [ "1. foo ", "foo" ]

re.lastIndex;           // 7--正确位置!
str.match( re );        // [ "2. bar ", "bar" ]

re.lastIndex;           // 14--正确位置!
str.match( re );        // ["3. baz", "baz"]
```

这种方式可以工作是因为我提前了解了输入字符串的结构：在想要匹配的内容（"foo" 等）之前总有一个像 "1. " 这样的数字前缀，然后其后或者是一个空格，或者是字符串结尾（也就是锚点 $）。所以我构造的正则表达式在主匹配中捕获所有这样的结构，然后使用匹配组 ( )，这样我真正关心的内容就很方便地被提取出来了。

第一次匹配（"1. foo"）之后，lastIndex 值为 7，这已经是下一个匹配 "2. bar" 开始所需的位置了，依次继续。

如果要使用 y 定点模式来重复匹配，你将很可能想要寻找像我们前面展示的自动更新 lastIndex 位置的机会。

2. 定点还是全局

有些读者可能意识到，也可以用 g 全局匹配标识和 exec(..) 方法来模拟这种相对于 lastIndex 的匹配，就像这样：

```
var re = /o+./g,        // <-- 注意g!
    str = "foot book more";

re.exec( str );        // ["oot"]
re.lastIndex;          // 4

re.exec( str );        // ["ook"]
re.lastIndex;          // 9

re.exec( str );        // ["or"]
re.lastIndex;          // 13

re.exec( str );        // null--没有更多匹配!
re.lastIndex;          // 0--现在从头开始!
```

确实，g 模式匹配加上 exec(..) 从 lastiIndex 的当前值开始匹配，同时会在每次匹配（或匹配失败后）更新 lastIndex，但是这和 y 的行为特性并不相同。

注意前面的代码中位于位置 6 处的 "ook"，会被第二个 exec(..) 调用匹配找到，虽然在那个时候，lastIndex 值是 4（来自于上一次匹配的结尾）。这是为什么？因为就像我前面所说，非定点匹配可以在匹配过程中自由向前移动。因为不允许向前移动，所以定点模式表达式在这里会匹配失败。

除了可能并不需要的向前移动匹配，只用 g 替代 y 的另一个缺点是 g 改变了某些匹配方法的行为特性，比如 str.match(re)。

考虑：

```
var re = /o+./g,        // <-- 注意g!
    str = "foot book more";

str.match( re );        // ["oot","ook","or"]
```

看到所有的匹配是如何立即返回了吗？有时候这不是问题，但有时候它又并非是你想要的。

通过使用工具 test(..) 和 match(..)，y 定点标识带来的是一次一个的向前匹配。只是要确保每次匹配的时候 lastIndex 总是处于正确的位置上！

3. 锚定

前面已经提醒过，把定点模式想象成就是从 ^ 开始的模式是不精确的。^ 锚点在正则表达式里有独特的含义，不会被定点模式所改变。^ 是一个总是指向输入起始处的锚点，和 lastIndex 完全没有任何关系。

除了一些贫乏 / 不精确文档的原因，实际上在 Firefox 上的一个老旧的前 ES6 定点模式试验确实让 ^ 与 lastIndex 相关，这也不幸地进一步加深了误解，所以这样的行为特性已经存在几年了。

ES6 选择不采取这种实现方式。模式中的 ^ 就是表示也仅表示输入的起始点。

因此，像 /^foo/y 这样的模式总是也只是寻找字符串起始处的 "foo" 匹配，但前提是允许在此处匹配。如果 lastIndex 不是 0，那么匹配就会失败。考虑：

```
var re = /^foo/y,
    str = "foo";

re.test( str );         // true
re.test( str );         // false

re.lastIndex;           // 0--失败后重置

re.lastIndex = 1;
re.test( str );         // false--由于定位而失败
re.lastIndex;           // 0--失败后重置
```

底线：y 加上 ^ 再加上 lastIndex > 0 是一个不兼容的组合，总是会导致匹配失败。

> y 不会以任何形式改变 ^ 的含义，而 m 多行模式则会，也就是说，^ 意味着输入起始处或者换行之后的文字起始位置。所以，如果组合使用 y 和 m 模式，就会发现字符串中有多个 ^ 匹配基准。但是记住：因为这是 y 定点的，需要确保每次匹配的时候 lastIndex 指向正确的换行位置（可能通过匹配行尾来实现），否则下次匹配不会成功。

2.10.3　正则表达式 flags

在 ES6 之前，如果想要通过检查一个正则表达式对象来判断它应用了那些标识，需要把它从 source 属性的内容中解析出来——讽刺的是，这可能需要用另一个正则表达式，就像下面这样：

```
var re = /foo/ig;

re.toString();          // "/foo/ig"

var flags = re.toString().match( /\/([gim]*)$/ )[1];

flags;                 // "ig"
```

而在 ES6 中，现在可以用新的 flags 属性直接得到这些值。

```
var re = /foo/ig;

re.flags;               // "gi"
```

这里有一点细节，ES6 规范中规定了表达式的标识按照这个顺序列出："gimuy"，无论原始指定的模式是什么。这就是 /ig 和 "gi" 之间区别的原因。

指定或列出的标识顺序是无关紧要的。

ES6 的另一个调整是如果把标识传给已有的正则表达式，`RegExp(..)` 构造器现在支持 `flags`：

```
var re1 = /foo*/y;
re1.source;                         // "foo*"
re1.flags;                          // "y"

var re2 = new RegExp( re1 );
re2.source;                         // "foo*"
re2.flags;                          // "y"

var re3 = new RegExp( re1, "ig" );
re3.source;                         // "foo*"
re3.flags;                          // "gi"
```

在 ES6 之前，`re3` 构造会抛出一个错误，但在 ES6 中，可以在复制的时候覆盖标识。

2.11　数字字面量扩展

在 ES5 之前，数字字面量看起来是这样的——八进制形式并没有被正式支持，只作为扩展支持，而浏览器已经达成实质上的一致：

```
var dec = 42,
    oct = 052,
    hex = 0x2a;
```

> 虽然可以用不同进制形式指定数字，但是数字的数字值还是保存的值，并且默认的输出解释形式总是十进制。前面代码中的 3 种形式在其中保存的都是值 `42`。

为了进一步展示 `052` 是一个非标准形式的扩展，考虑：

```
Number( "42" );             // 42
Number( "052" );            // 52
Number( "0x2a" );           // 42
```

ES5 仍然支持作为浏览器扩展的八进制形式（包括这样的不一致），除了在严格模式下，是不支持八进制字面量（`052`）形式的。采用这个限制主要是因为很多开发者有这样一种（来自于其他语言经验的）习惯，就是看起来无意地在十进制值前加 `0` 用于代码对齐，然后就会发现已经不小心完全改变了这个数值！

在非十进制数字字面量表示方面，ES6 继续支持旧有的修改 / 变体。同时现在有了一个正式八进制形式、一个补充的十六进制形式，以及一个全新的二进制形式。由于 Web 兼容性

的原因，旧有的八进制 052 形式在非严格模式下还是合法的（尽管没有指出），但不应该再使用这种形式了。

下面是新的 ES6 数字字面量形式：

```
var dec = 42,
    oct = 0o52,          // 或者0O52 :(
    hex = 0x2a,          // 或者0X2a :/
    bin = 0b101010;      // 或者0B101010 :/
```

唯一合法的小数形式是十进制的。八进制、十六进制和二进制都是整数形式。

这些形式的字符串表示都可以强制类型转换 / 变换成相应的数字值：

```
Number( "42" );          // 42
Number( "0o52" );        // 42
Number( "0x2a" );        // 42
Number( "0b101010" );    // 42
```

尽管并非是 ES6 中全新的，但有一个不为人知的事实就是这些形式都可以进行（某种程度的）反向转换：

```
var a = 42;

a.toString();            // "42"--也可以用a.toString( 10 )
a.toString( 8 );         // "52"
a.toString( 16 );        // "2a"
a.toString( 2 );         // "101010"
```

实际上，可以用这种方式以任何 2 到 36 之间的基表示一个数字，虽然像 2、8、10 和 16 这些标准基范围之外的表示法非常少见。

2.12　Unicode

先声明这一小节并非是对 Unicode 资源完整详尽的介绍。我将会覆盖 ES6 中你需要了解的关于 Unicdoe 的**变化**，但也不会比这更深入太多了。关于 JavaScript 和 Unicode，Mathias Bynens（http://twitter.com/ mathias）编写 / 发表了详尽睿智的介绍（参考 https://mathiasbynens.be/notes/javascript-unicode 和 http:// fluentconf.com/javascript-html-2015/public/content/ 2015/02/18-javascript-loves-unicode）。

Unicode 字符范围从 0x0000 到 0xFFFF，包含可能看到和接触到的所有（各种语言的）标准打印字符。这组字符称为**基本多语言平面**（Basic Multilingual Plane，BMP）。BMP 甚至包含了像雪人这样的有趣的符号：☃（U+2603）。

在 BMP 集之外还有很多其他扩展 Unicode 字符，范围直到 0x10FFFF。这些符号通常是**星形符号**（astral symbol），这个名称是指 BMP 之外的字符的 16 个**平面**（或者说，层次 / 分

组）的集合。星形符号的例子包括𝄞 U+1D11E）和💩 U+1F4A9) 这样的符号。

在 ES6 之前，可以通过 Unicode 转义符指定 JavaScript 字符串为 Unicode 字符，比如：

```
var snowman = "\u2603";
console.log( snowman );       // "☃"
```

但是，Unicode 转义符 `\uXXXX` 只支持四个十六进制字符的形式，所以这种方法只能表示 BMP 字符集。要在 ES6 之前用 Unicode 转义符表示 astral 字符，需要使用一个**替代对**——简单说就是连续两个特别计算出来的 Unicode 转义字符，JavaScript 解释器会把它解释为单个 astral 字符：

```
var gclef = "\uD834\uDD1E";
console.log( gclef );       // "𝄞"
```

而在 ES6 中，现在有了可以用于作 Unicode 转义（在字符串和正则表达式中）的新形式，称为 Unicode **码点转义**（code point escaping）：

```
var gclef = "\u{1D11E}";
console.log( gclef );       // "𝄞"
```

你可以看到，区别在于转义序列中出现了 `{ }`，支持在其中包含任意多个十六进制字符。因为最多只需要六个就可以表示 Unicode 中最大的可能码点（也就是 `0x10FFFF`），所以这足够了。

2.12.1 支持 Unicode 的字符串运算

默认情况下，JavaScript 字符串运算和方法不能感知字符串中的 astral 符号。所以会单独处理每个 BMP 字符，即使是构成单个 astral 字符的两半。考虑：

```
var snowman = "☃";
snowman.length;                 // 1

var gclef = "𝄞";
gclef.length;                   // 2
```

那么如何精确计算这样的字符串的长度呢？在这种情况下，可以使用下面的技巧：

```
var gclef = "𝄞";

[...gclef].length;              // 1
Array.from( gclef ).length;     // 1
```

回忆本章前面 2.9 节所介绍的，ES6 字符串有内建的迭代器。这个迭代器恰好是可以识别 Unicode 的，也就是说它能够自动将 astral 符号作为单个值输出。我们可以利用这一点，在数组字面量使用 `...`spread 运算符，创建一个字符串符号的数组。然后查看结果数组的长

度。ES6 的 `Array.from(..)` 所做的事情基本上和 `[...XYZ]` 一样，我们将在第 6 章详细介绍这个工具。

> 应该注意，与理论上优化过的原生工具 / 属性的做法相比，只是为了得到一个字符串的长度就需要构造并消耗一个迭代器，这种方法的性能代价相对来说是非常昂贵的。

不幸的是，完整的答案并没有那么简单直接。除了替代字符对（字符串迭代器会处理），还有特殊 Unicode 码点需要用特殊的方式处理，这就更难计算了。例如，有一组码点会修改前面相邻的字符，被称为**组合音标符号**（Combining Diacritical Mark）。

考虑这两个字符串输出：

```
console.log( s1 );              // "é"
console.log( s2 );              // "é"
```

看起来是一样的，但实际上并非如此！下面是创建 `s1` 和 `s2` 的过程：

```
var s1 = "\xE9",
    s2 = "e\u0301";
```

可能你已经猜到，前面的 `length` 计算技巧对于 `s2` 并不适用：

```
[...s1].length;                 // 1
[...s2].length;                 // 2
```

那么应该怎么办呢？这种情况下，可以在查询长度之前使用 ES6 的 `String#normalize(..)` 工具（我们将在第 6 章进一步介绍）对这个值执行 **Unicode 规范化**（Unicode normalization）：

```
var s1 = "\xE9",
    s2 = "e\u0301";

s1.normalize().length;         // 1
s2.normalize().length;         // 1

s1 === s2;                     // false
s1 === s2.normalize();         // true
```

基本上说就是，`normalize(..)` 接受像 `"e\u0301"` 这样的一个序列，然后把它规范化为 `"\xE9"`。甚至如果有合适的 Unicode 符号可以合并的话，规范化可以把多个相邻的组合符号合并：

```
var s1 = "o\u0302\u0300",
    s2 = s1.normalize(),
    s3 = "ồ";

s1.length;                    // 3
```

```
s2.length;                    // 1
s3.length;                    // 1

s2 === s3;                    // true
```

不幸的是，这里规范化也并不完美。如果有多个组合符号修改了单个字符，你可能就无法得到期望的长度结果，因为可能并没有单个的定义好的规范化符号可以表示所有这些符号带来的合并结果。举例来说：

```
var s1 = "e\u0301\u0330";

console.log( s1 );             // "ḛ́"

s1.normalize().length;         // 2
```

你越是深入挖掘这个兔子洞，就越会意识到很难得到一个对“长度”的精确定义。我们视觉上看到的渲染出来的单个字符——更精确的叫法是**字素**（grapheme）——并不总是与程序处理意义上的单个“字符”严格对应。

> 如果你想了解这个兔子洞到底有多深，参见“字素族界限”算法（http://www.Unicode.org/reports/tr29/#Grapheme_Cluster_Boundaries）。

2.12.2 字符定位

与长度复杂性类似，“位于位置 2 处的字符是什么？”的精确含义又是什么呢？原生的前 ES6 对此的答案是 `charAt(..)`，但它不支持 astral 字符的原子性，也不会考虑组合符号的因素。

考虑：

```
var s1 = "abc\u0301d",
    s2 = "ab\u0107d",
    s3 = "ab\u{1d49e}d";

console.log( s1 );                // "abćd"
console.log( s2 );                // "abćd"
console.log( s3 );                // "ab𝒞d"

s1.charAt( 2 );                   // "c"
s2.charAt( 2 );                   // "ć"
s3.charAt( 2 );                   // "" <-- 不可打印
s3.charAt( 3 );                   // "" <-- 不可打印
```

那么 ES6 是否给出了支持 Unicode 版本的 `charAt(..)` 呢？不幸的是，并没有。但在编写本部分时，已经有一个这样的后 ES6 提案在考虑之中了。

但有了前一小节中介绍的内容（当然也包括给出提醒的局限性！），我们可以给出 ES6 版本的回答：

```
var s1 = "abc\u0301d",
    s2 = "ab\u0107d",
    s3 = "ab\u{1d49e}d";

[...s1.normalize()][2];         // "ć"
[...s2.normalize()][2];         // "ć"
[...s3.normalize()][2];         // "𝒞"
```

记住前面的提醒：从性能的角度看，每次想要得到单个字符都要构造并消耗一个迭代器是……非常不理想的。我们期待后 ES6 优化的内建工具尽快出现。

那么支持 Unicode 的版本工具 `charCodeAt(..)` 怎么样呢？ ES6 提供了 `codePointAt(..)`：

```
var s1 = "abc\u0301d",
    s2 = "ab\u0107d",
    s3 = "ab\u{1d49e}d";

s1.normalize().codePointAt( 2 ).toString( 16 );
// "107"

s2.normalize().codePointAt( 2 ).toString( 16 );
// "107"

s3.normalize().codePointAt( 2 ).toString( 16 );
// "1d49e"
```

反方向呢？ ES6 中支持 Unicode 版本的 `String.fromCharCode(..)` 是 `String.fromCodePoint(..)`：

```
String.fromCodePoint( 0x107 );       // "ć"

String.fromCodePoint( 0x1d49e );     // "𝒞"
```

那么稍等，能不能组合 `String.fromCodePoint(..)` 和 `codePointAt(..)` 来获得支持 Unicode 的 `charAt(..)` 的更简单且更优的方法呢？是的，能！

```
var s1 = "abc\u0301d",
    s2 = "ab\u0107d",
    s3 = "ab\u{1d49e}d";

String.fromCodePoint( s1.normalize().codePointAt( 2 ) );
// "ć"

String.fromCodePoint( s2.normalize().codePointAt( 2 ) );
// "ć"
```

```
String.fromCodePoint( s3.normalize().codePointAt( 2 ) );
// "𝒞"
```

还有很多字符串方法这里没有介绍，包括 `toUpperCase()`、`toLowerCase()`、`substring(..)`、`indexOf(..)`、`slice(..)`，以及几十个其他方法。这些方法都没有修改或增补以提供完整的 Unicode 支持，所以在处理包含 astral 符号的字符串的时候应该非常小心——还是尽可能避免使用吧！

也有一些字符串方法使用了正则表达式实现其功能，比如 `replace(..)` 和 `match(..)`。谢天谢地，正如我们在 2.10.1 节中介绍的，ES6 为正则表达式提供了 Unicode 支持。

好吧，现在可以了！通过前面介绍的各种新增特性可以看出，JavaScript 的 Unicode 字符串支持明显优于前 ES6 版本（尽管还不完美）。

2.12.3 Unicode 标识符名

Unicode 也可以用作标识符名（变量、属性等）。在 ES6 之前，可以通过 Unicode 转义符实现这一点，比如：

```
var \u03A9 = 42;

// 等价于：var Ω = 42;
```

而在 ES6 中，还可以使用前面解释过的码点转义符语法：

```
var \u{2B400} = 42;

// 等价于：var 𫐀 = 42;
```

关于到底支持哪些 Unicode 字符，有一套复杂的规则。另外，还有一些只允许出现在标识符名称中的非首个字符位置上。

> 有关所有这些细节，Mathias Bynens 写了一篇很好的文章（https://mathiasbynens.be/notes/javascript-identifiers-es6）。

在标识符名称中使用这些不常见字符的原因是很罕见的，也只是学术意义上的。通常最好不要编写依赖于这些晦涩特性的代码。

2.13 符号

ES6 为 JavaScript 引入了一个新的原生类型：`symbol`，这是很久没有发生过的事情。但是，和其他原生类型不一样，symbol 没有字面量形式。

下面是创建 symbol 的过程：

```
var sym = Symbol( "some optional description" );

typeof sym;     // "symbol"
```

以下几点需要注意。

- 不能也不应该对 `Symbol(..)` 使用 `new`。它并不是一个构造器，也不会创建一个对象。
- 传给 `Symbol(..)` 的参数是可选的。如果传入了的话，应该是一个为这个 symbol 的用途给出用户友好描述的字符串。
- `typeof` 的输出是一个新的值 (`"symbol"`)，这是识别 symbol 的首选方法。

如果提供了描述的话，它只被用作为这个符号的字符串表示：

```
sym.toString();     // "Symbol(some optional description)"
```

如同原生字符串值不是 `String` 的实例一样，symbol 也不是 `Symbol` 的实例。如果出于某种原因想要构造一个 symbol 值的装箱封装对象形式，可以使用下面的方法：

```
sym instanceof Symbol;      // false

var symObj = Object( sym );
symObj instanceof Symbol;   // true

symObj.valueOf() === sym;   // true
```

> 这段代码中的 `symObj` 也可以换作 `sym`；两种形式都在所有使用 symbol 的场合适用。需要使用装箱封装对象形式（`symObj`）而不是原生形式（`sym`）的情况很少。和针对其他原生类型的建议一样，最好使用 `sym` 代替 `symObj`。

符号本身的内部值——称为它的名称（`name`）——是不在代码中出现且无法获得的。可以把这个符号值想象为一个自动生成的、(在应用内部）唯一的字符串值。

但如果这个值是隐藏的且无法获得，那么符号的存在意义是什么呢？

符号的主要意义是创建一个类 (似) 字符串的不会与其他任何值冲突的值。所以，考虑使用一个符号作为事件名的常量表示的例子：

```
const EVT_LOGIN = Symbol( "event.login" );
```

然后在需要像 `"event.login"` 这样的一般字符串字面量的地方就可以使用 `EVT_LOGIN` 了：

```
evthub.listen( EVT_LOGIN, function(data){
```

```
    // ..
} );
```

这里的好处是 EVT_LOGIN 持有一个不可能与其他值（有意或无意）重复的值，所以这里分发或处理的事件不会有任何混淆。

> 在底层，前面代码中的 evthub 工具很可能在某个用来跟踪事件处理函数的内部对象（hash）中直接使用 EVT_LOGIN 参数的符号值属性 / 键值。如果 evthub 需要把这个符号值作为一个真正的字符串使用，那么它会需要用 String() 或者 toString() 进行显式类型转换，因为不允许隐式地把符号转换为字符串。

可以在对象中直接使用符号作为属性名 / 键值，比如用作一个特殊的想要作为隐藏或者元属性的属性。尽管通常会这么使用，但是它**实际上**并不是隐藏的或者无法接触的属性，了解这一点很重要。

考虑一下这个实现了**单例**（singleton）模式的模块，也就是说，它只允许自己被创建一次：

```
const INSTANCE = Symbol( "instance" );

function HappyFace() {

    if (HappyFace[INSTANCE]) return HappyFace[INSTANCE];

    function smile() { .. }

    return HappyFace[INSTANCE] = {
        smile: smile
  };
}

var me = HappyFace(),
    you = HappyFace();

me === you;                      // true
```

这里的 INSTANCE 符号值是一个特殊的、几乎隐藏的、类似元属性的属性，静态保存在 HappeyFace() 函数对象中。

也可以采用平凡的、旧有的属性，比如 __instance，其行为特性是相同的。符号的使用只是改进了元编程风格，把 INSTANCE 属性与其他普通属性区分开来。

2.13.1　符号注册

在后几个例子中使用符号有几个细微缺点，EVT_LOGIN 和 INSTANCE 变量不得不保存在外层作用域中（可能甚至是全局作用域），或者保存在某个公开可用的位置，这样所有需要使

用这些符号的代码才都能够访问它们。

要改进访问这些符号的代码的组织形式，可以通过**全局符号注册**（global symbol registry）创建这些符号值。举例来说：

```
const EVT_LOGIN = Symbol.for( "event.login" );

console.log( EVT_LOGIN );       // Symbol(event.login)
```

以及

```
function HappyFace() {
    const INSTANCE = Symbol.for( "instance" );

    if (HappyFace[INSTANCE]) return HappyFace[INSTANCE];

    // ..

    return HappyFace[INSTANCE] = { .. };
}
```

`Symbol.for(..)` 在全局符号注册表中搜索，来查看是否有描述文字相同的符号已经存在，如果有的话就返回它。如果没有的话，会新建一个并将其返回。换句话说，全局注册表把符号值本身根据其描述文字作为单例处理。

但是，这也意味着只要使用的描述名称匹配，可以在应用的任何地方通过 `Symbol.for(..)` 从注册表中获取这个符号。

具有讽刺意义的是，基本上符号的目的是为了取代应用中的 **magic 字符串**（magic string，赋予特殊意义的任意字符串）。但在全局符号注册表中恰恰是用 magic 字符串值来唯一标识 / 定位符号。

为了避免意外冲突，可能需要符号描述唯一。一个简单的实现方法是在其中包含前缀 / 上下文 / 名字空间信息。

例如，考虑下面这个工具：

```
function extractValues(str) {
    var key = Symbol.for( "extractValues.parse" ),
        re = extractValues[key] ||
            /[^=&]+?=([^&]+?)(?=&|$)/g,
        values = [], match;

    while (match = re.exec( str )) {
        values.push( match[1] );
}

    return values;
}
```

这里使用了 "extractValues.parse" 这个 magic 字符串值，因为注册表中其他符号的描述与之冲突的可能性不大。

如果这个工具的用户想要覆盖解析正则表达式，也可以使用符号注册：

```
extractValues[Symbol.for( "extractValues.parse" )] =
    /..some pattern../g;

extractValues( "..some string.." );
```

符号注册为这些值提供了全局存储，除了这个帮助之外，这里看到的所有示例实际上都可以通过直接用 magic 字符串 "extractValues.parse"，而不是符号作为键值来实现。其改进更多是在元编程这一层次上而不是在函数这一层。

可以使用已经存储在注册中的符号值寻找其底层存储的描述文本（键值）。比如，因为无法传递符号本身，可能需要向应用的另外一部分发送信号告诉它如何在注册表中定位这个符号。

可以使用 Symbol.keyFor(..) 提取注册符号的描述文本（键值）：

```
var s = Symbol.for( "something cool" );

var desc = Symbol.keyFor( s );
console.log( desc );            // "something cool"

// 再次从注册中取得符号
var s2 = Symbol.for( desc );

s2 === s;                       // true
```

2.13.2　作为对象属性的符号

如果把符号用作对象的属性 / 键值，那么它会以一种特殊的方式存储，使得这个属性不出现在对这个对象的一般属性枚举中：

```
var o = {
    foo: 42,
    [ Symbol( "bar" ) ]: "hello world",
    baz: true
};

Object.getOwnPropertyNames( o );    // [ "foo","baz" ]
```

要取得对象的符号属性：

```
Object.getOwnPropertySymbols( o );  // [ Symbol(bar) ]
```

这就很清楚地表明了，属性符号实际上并不是隐藏或不可访问的，因为总可以通过 `Object.getOwnPropertySymbols(..)` 列表看到它。

内置符号

ES6 支持若干预先定义好的内置符号，它们可以暴露 JavaScript 对象值的各种元特性。但是，这些符号**并不是**像一般设想的那样注册在全局符号表里。

相反，它们作为 `Symbol` 函数对象的属性保存。比如 2.9 节中介绍的 `Symbol.iterator` 值：

```
var a = [1,2,3];

a[Symbol.iterator];          // 原生函数
```

规范使用 `@@` 前缀记法来指代内置符号，最常用的一些是：`@@iterator`、`@@toStringTag`、`@@toPrimitive`。规范还定义了一些其他符号，但是可能没那么常用。

> 关于如何在元编程中应用这些内置符号，参见 7.3 节。

2.14 小结

ES6 为 JavaScript 增加了很多新的语法形式，所以要学的太多了！

这些新语法形式中大多数的设计目的都是消除常见编程技巧中的痛点，比如为函数参数设定默认值以及把参数的“其余”部分收集到数组中。解构是一个强有力的工具，用于更精确地表达从数组和嵌套对象中赋值。

而像 `=>` 箭头函数这样的特性看起来似乎是为了使代码更简洁的语法，但实际上它有非常特别的行为特性，应该只在适当的时候使用。

扩展 Unicode 支持、新的正则表达式技巧，甚至新的基本类型 `symbol` 都使 ES6 的语法发展的更加完善。

第 3 章

代码组织

编写 JavaScript 代码是一回事，而合理组织代码则是另一回事。利用通用模式来组织和复用代码显著提高了代码的可读性和可理解性。记住：对于代码来说，和其他开发者交流与提供计算机指令同等重要。

ES6 提供了几个重要的特性，显著改进了以下模式，包括迭代器、生成器、模块和类。

3.1 迭代器

迭代器（iterator）是一个结构化的模式，用于从源以一次一个的方式提取数据。这个模式在编程中已经使用相当长的一段时间了。从很久之前开始，JavaScript 开发者就已经在 JavaScript 程序中自发地设计和实现迭代器，所以这不是一个全新的主题。

ES6 实现的是为迭代器引入一个隐式的标准化接口。JavaScript 很多内建的数据结构现在都提供了实现这个标准的迭代器。为了达到最大化的互操作性，也可以自己构建符合这个标准的迭代器。

迭代器是一种有序的、连续的、基于拉取的用于消耗数据的组织方式。

例如，你可以实现一个工具，在每次请求的时候产生一个新的唯一标识符。也可以在一个固定列表上以轮询的方式产生一个无限值序列。或者也可以把迭代器附着在一个数据库查询结果上，每次迭代拉出一个新行。

虽然在 JavaScript 中这样的用法并不常见，但是迭代器也可以用于以一次一步的方式控制

行为。将生成器（参见 3.2 节）考虑在内，可以把这一点展示的非常清晰，尽管不用生成器也完全可以实现这一功能。

3.1.1 接口

在编写本部分的时候，ES6 第 25.1.1.2 节（https:// people.mozilla.org/~jorendorff/es6-draft.html#sec-iterator-interface）详细解释了 `Iterator` 接口，包括如下要求：

```
Iterator [required]
    next() {method}: 取得下一个IteratorResult
```

有些迭代器还扩展支持两个可选成员：

```
Iterator [optional]
    return() {method}: 停止迭代器并返回IteratorResult
    throw() {method}: 报错并返回IteratorResult
```

`IteratorResult` 接口指定如下：

```
IteratorResult
    value {property}: 当前迭代值或者最终返回值（如果undefined为可选的）
    done {property}: 布尔值，指示完成状态
```

> 我把这些接口称为隐式的，并不是因为它们没有在规范中显式说明——它们有说明！——而是因为它们没有暴露为代码可以访问的直接对象。在 ES6 中，JavaScript 不支持任何“接口”的概念，所以你自己的代码符合规范只是单纯的惯用法。然而，在 JavaScript 期望迭代器的位置 (比如 `for..of` 循环) 所提供的东西必须符合这些接口，否则代码会失败。

还有一个 `Iterable` 接口，用来表述必需能够提供生成器的对象：

```
Iterable
    @@iterator() {method}: 产生一个 Iterator
```

回忆一下 2.13.2 节，你会了解到 `@@iterator` 是一个特殊的内置符号，表示可以为这个对象产生迭代器的方法。

`IteratorResult`

`IteratorResult` 接口指定了从任何迭代器操作返回的值必须是下面这种形式的对象：

```
{ value: .. , done: true / false }
```

内置迭代器总是返回这种形式的值，当然如果需要的话，返回值还可以有更多的属性。

例如，自定义迭代器可能在结果对象上增加额外的元数据（比如数据的来源、获取数据的

时间长度、缓存过期时长、下次请求的适当频率，等等）。

> 严格来说，如果不提供 value 可以被当作是不存在或者未设置，就像值 undefined，那么 value 是可选的。因为访问 res.value 的时候，不管它存在且值为 undefined，还是根本不存在，都会产生 undefined，这个属性的存在 / 缺席更多的是一个实现细节或者优化技术（或者二者兼有），而非功能问题。

3.1.2　next() 迭代

我们来观察一个数组，这是一个 iterable，它产生的迭代器可以消耗其自身值：

```
var arr = [1,2,3];

var it = arr[Symbol.iterator]();

it.next();      // { value: 1, done: false }
it.next();      // { value: 2, done: false }
it.next();      // { value: 3, done: false }

it.next();      // { value: undefined, done: true }
```

每次在这个 arr 值上调用位于 Symbol.iterator（参见第 2 章和第 7 章）的方法时，都会产生一个全新的迭代器。多数结构都是这么实现的，包括所有 JavaScript 内置数据结构。

但像事件队列消费者这样的结构可能只产生一个迭代器（单例模式）。或者某个结构可能在同一时刻只允许唯一的迭代器，要求当前的迭代器完成才能创建下一个迭代器。

前面代码中在提取值 3 的时候，迭代器 it 不会报告 done: true。必须得再次调用 next()，越过数组结尾的值，才能得到完成信号 done: true。虽然直到本小节的后面才会介绍原因，但这样的设计决策通常被认为是最佳实践。

基本字符串值默认也可以迭代：

```
var greeting = "hello world";

var it = greeting[Symbol.iterator]();

it.next();      // { value: "h", done: false }
it.next();      // { value: "e", done: false }
..
```

> 严格来说，基本值本身不是 iterable，但是感谢“封箱”技术，"hello world" 被强制转换 / 变换为 String 对象封装形式，而这是一个 iterable。参见本系列《你不知道的 JavaScript（中卷）》第一部分可以获取更多细节。

ES6 中还包括几个新的称为集合（参见第 5 章）的数据结构。这些集合不仅本身是 iterable，还提供了 API 方法来产生迭代器，比如：

```
var m = new Map();
m.set( "foo", 42 );
m.set( { cool: true }, "hello world" );

var it1 = m[Symbol.iterator]();
var it2 = m.entries();

it1.next();     // { value: [ "foo", 42 ], done: false }
it2.next();     // { value: [ "foo", 42 ], done: false }
..
```

迭代器的 `next(..)` 方法可以接受一个或多个可选参数。绝大多数内置迭代器没有利用这个功能，尽管生成器的迭代器肯定有（参见 3.2 节）。

通用的惯例是，包括所有内置迭代器，在已经消耗完毕的迭代器上调用 `next(..)` 不会出错，而只是简单地继续返回结果 `{ value: undefined, done: true }`。

3.1.3 可选的 `return(..)` 和 `throw(..)`

多数内置迭代器都没有实现可选的迭代器接口——`return(..)` 和 `throw(..)`。然而，在生成器的上下文中它们肯定是有意义的，参见 3.2 节获取更多信息。

`return(..)` 被定义为向迭代器发送一个信号，表明消费者代码已经完毕，不会再从其中提取任何值。这个信号可以用于通知生产者（响应 `next(..)` 调用的迭代器）执行可能需要的清理工作，比如释放 / 关闭网络、数据库或者文件句柄资源。

如果迭代器存在 `return(..)`，并且出现了任何可以自动被解释为异常或者对迭代器消耗的提前终止的条件，就会自动调用 `return(..)`。你也可以手动调用 `return(..)`。

`return(..)` 就像 `next(..)` 一样会返回一个 `IteratorResult` 对象。一般来说，发送给 `return(..)` 的可选值将会在这个 `IteratorResult` 中作为 `value` 返回，但在一些微妙的情况下并非如此。

`throw(..)` 用于向迭代器报告一个异常 / 错误，迭代器针对这个信号的反应可能不同于针对 `return(..)` 意味着的完成信号。和对于 `return(..)` 的反应不一样，它并不一定意味着迭代器的完全停止。

例如，通过生成器迭代器，`throw(..)` 实际上向生成器的停滞执行上下文中插入了一个抛出的异常，这个异常可以用 `try..catch` 捕获。未捕获的 `throw(..)` 异常最终会异常终止生成器迭代器。

通用的惯例是，迭代器不应该在调用 return(..) 或者 thrown(..) 之后再产生任何值。

3.1.4 迭代器循环

我们在 2.9 节已经介绍过，ES6 的 for..of 循环直接消耗一个符合规范的 iterable。

如果一个迭代器也是一个 iterable，那么它可以直接用于 for..of 循环。你可以通过为迭代器提供一个 Symbol.iterator 方法简单返回这个迭代器本身使它成为 iterable：

```
var it = {
    // 使迭代器it成为iterable
    [Symbol.iterator]() { return this; },

    next() { .. },
    ..
};

it[Symbol.iterator]() === it;     // true
```

现在可以用 for..of 循环消耗这个 it 迭代器：

```
for (var v of it) {
    console.log( v );
}
```

要彻底理解这样的循环如何工作，可以回顾一下第 2 章 for..of 循环的等价 for 形式：

```
for (var v, res; (res = it.next()) && !res.done; ) {
    v = res.value;
    console.log( v );
}
```

如果认真观察的话，可以看到每次迭代之前都调用了 it.next()，然后查看一下 res.done。如果 res.done 为 true，表达式求值为 false，迭代就不会发生。

回忆一下，前面我们建议迭代器一般不应与最终预期的值一起返回 done: true。现在能明白其中的原因了吧。

如果迭代器返回 { done: true, value: 42 }，for..of 循环会完全丢弃值 42，那么这个值就被丢失了。因为这个原因，假定你的迭代器可能会通过 for..of 循环或者手动的等价 for 形式模式消耗，那么你应该等返回所有的相关迭代值之后，再返回 done: true 来标明迭代完毕。

> 当然，可以有意地把迭代器设计为在返回 `done: true` 的同时返回一些相关值。但除非已经写了文档表明这一点，以此迫使迭代器的消费者使用不同的迭代模式，而不是像 `for..of` 或者其等价手动 `for` 形式暗示的那样，否则不要这么做。

3.1.5　自定义迭代器

除了标准的内置迭代器，你也可以构造自己的迭代器！要使得它们能够与 ES6 的消费者工具（比如，`for..of` 循环以及 `...` 运算符）互操作，所需要做的就是使其遵循适当的接口。

让我们试着构造一个迭代器来产生一个无限斐波纳契序列：

```
var Fib = {
    [Symbol.iterator]() {
        var n1 = 1, n2 = 1;

        return {
            // 使迭代器成为iterable
            [Symbol.iterator]() { return this; },

            next() {
                var current = n2;
                n2 = n1;
                n1 = n1 + current;
                return { value: current, done: false };
            },

            return(v) {
                console.log(
                    "Fibonacci sequence abandoned."
                );
                return { value: v, done: true };
            }
        };
    }
};

for (var v of Fib) {
    console.log( v );

    if (v > 50) break;
}
// 1 1 2 3 5 8 13 21 34 55
// Fibonacci sequence abandoned.
```

> 如果我们没有插入 `break` 条件的话，这个 `for..of` 循环就会无限循环下去，这可能不是你想要的结果。

调用 Fib[Symbol.iterator]() 方法的时候，会返回带有 next() 和 return(..) 方法的迭代器对象。通过放在闭包里的变量 n1 和 n2 维护状态。

接下来考虑一个迭代器，它的设计意图是用来在一系列（也就是一个队列）动作上运行，一次一个条目：

```
var tasks = {
    [Symbol.iterator]() {
        var steps = this.actions.slice();

        return {
            // 使迭代器成为iterable
            [Symbol.iterator]() { return this; },

            next(...args) {
                if (steps.length > 0) {
                   let res = steps.shift()( ...args );
                   return { value: res, done: false };
                }
                else {
                    return { done: true }
                }
            },

            return(v) {
                steps.length = 0;
                return { value: v, done: true };
            }
        };
    },
    actions: []
};
```

tasks 上的迭代器走过 actions 数组属性中找到的函数（如果有的话），然后一次一个执行这些函数，把传入 next(..) 的所有参数传入，将其返回值在标准 IteratorResult 对象中返回。

下面是这个 tasks 队列的一种使用方式：

```
tasks.actions.push(
    function step1(x){
        console.log( "step 1:", x );
        return x * 2;
    },
    function step2(x,y){
        console.log( "step 2:", x, y );
        return x + (y * 2);
    },
    function step3(x,y,z){
        console.log( "step 3:", x, y, z );
        return (x * y) + z;
```

```
    }
);

var it = tasks[Symbol.iterator]();

it.next( 10 );          // step 1: 10
                        // { value: 20, done: false }

it.next( 20, 50 );      // step 2: 20 50
                        // { value:  120, done: false }

it.next( 20, 50, 120 ); // step 3: 20 50 120
                        // { value: 1120, done: false }

it.next();              // { done: true }
```

这种特定的用法强调了迭代器可以作为一个模式来组织功能，而不仅仅是数据。下一小节我们介绍生成器时也可以回顾一下这里。

你甚至可以创造性地定义一个迭代器来表示单个数据上的元操作。举例来说，我们可以为数字定义一个迭代器，默认范围是从 0 到（或者对于负数来说，向下到）关注的数字。

考虑：

```
if (!Number.prototype[Symbol.iterator]) {
    Object.defineProperty(
        Number.prototype,
        Symbol.iterator,
        {

            writable: true,
            configurable: true,
            enumerable: false,
            value: function iterator(){
                var i, inc, done = false, top = +this;

                // 正向还是反向迭代?
                inc = 1 * (top < 0 ? -1 : 1);

                return {
                    // 使得迭代器本身成为iterable!
                    [Symbol.iterator](){ return this; },

                    next() {
                        if (!done) {
                            // 初始迭代总是0
                            if (i == null){
                                i = 0;
                            }
                            // 正向迭代
                            else if (top >= 0) {
                                i = Math.min(top,i + inc);
                            }
```

```
                            // 反向迭代
                            else {
                                i = Math.max(top,i + inc);
                            }

                            // 本次迭代后结束？
                            if (i == top) done = true;

                            return { value: i, done: false };
                        }
                        else {
                            return { done: true };
                        }
                    }
                };
            }
        }
    );
}
```

那么这种创造性提供了哪些技巧呢？

```
for (var i of 3) {
    console.log( i );
}
// 0 1 2 3

[...-3];                // [0,-1,-2,-3]
```

这是一些有趣的技巧，尽管其实际功效值得商榷。但又一次地，有人可能会奇怪为什么 ES6 不把这个微小但可爱的功能作为复活节彩蛋提供呢！

这一点我不能疏于提醒，像我在前面代码中那样扩展原生原型需要格外小心和清醒，以避免隐患。

在这个例子中，与其他代码甚至是未来的 JavaScript 功能冲突的可能性是微乎其微的。但要清醒意识到这么一丝的可能性。还要编写文档为后来者解释你的所作所为。

> 如果你想了解更多细节，在这篇博客文章（http://blog.getify.com/iterating-es6-numbers/）里我已经解释过这种特殊技术。还有这篇评论（http://blog.getify.com/iterating-es6-numbers/comment-page-1/#comment-535294）甚至建议为字符范围构造类似的技巧。

3.1.6　迭代器消耗

前面已经展示了如何通过 `for..of` 循环一个接一个地消耗迭代器项目，但是还有其他 ES6 结构可以用来消耗迭代器。

考虑一下附着在这个数组上的迭代器（尽管任何迭代器都有如下性质）：

```
var a = [1,2,3,4,5];
```

spread 运算符 `...` 完全消耗了迭代器。考虑：

```
 function foo(x,y,z,w,p) {
    console.log( x + y + z + w + p );
}

foo( ...a );            // 15
```

`...` 也可以把一个迭代器展开到一个数组中：

```
var b = [ 0, ...a, 6 ];
b;                      // [0,1,2,3,4,5,6]
```

数组解构（参见 2.4 节）可以部分或完全（如果和 rest / gather 运算符 `...` 配对使用的话）消耗一个迭代器：

```
var it = a[Symbol.iterator]();

var [x,y] = it;
// 从it中获取前两个元素
var [z, ...w] = it;
// 获取第三个元素，然后一次取得其余所有元素

// it已经完全耗尽？是的。
it.next();              // { value: undefined, done: true }

x;                      // 1
y;                      // 2
z;                      // 3
w;                      // [4,5]
```

3.2　生成器

所有的函数都运行直到完毕，对吗？换句话说，一旦一个函数开始运行，在它结束之前不会被任何事情打断。

至少对于 JavaScript 到目前为止的整个历史来说，是这样的。而 ES6 引入了一个全新的某种程度上说是奇异的函数形式，称为生成器。生成器可以在执行当中暂停自身，可以立即恢复执行也可以过一段时间之后恢复执行。所以显然它并不像普通函数那样保证运行到完毕。

还有，在执行当中的每次暂停 / 恢复循环都提供了一个双向信息传递的机会，生成器可以返回一个值，恢复它的控制代码也可以发回一个值。

和前一节的迭代器一样，可以从多个角度理解生成器是什么，或者最适合做什么。没有单

个正确答案，我们是试着从几个角度考虑的。

> 参见本系列《你不知道的 JavaScript（中卷）》第二部分可以获取关于生成器的更多信息，也可以参考第 4 章同名小节。

3.2.1　语法

通过以下新语法声明生成器函数：

```
function *foo() {
    // ..
}
```

从功能上来说，* 的位置无所谓。同样的声明可以写作：

```
function *foo()  { .. }
function* foo()  { .. }
function * foo() { .. }
function*foo()   { .. }
..
```

这里的**唯一**区别就是风格喜好。多数其他文献似乎都喜爱 `function* foo(..) { .. }` 这种形式。但我喜爱 `function *foo(..) { .. }`，所以后面章节都会采用这种形式。

我的理由纯粹就是说教性质的。本部分中，当提到生成器函数时，我都会使用 `*foo(..)`，而用 `foo(..)` 来指代普通函数。我发现 `*foo(..)` 与 `function *foo(..) { .. }` 中 * 的位置更加吻合。

还有，正如我们在第 2 章中已经看到的简洁方法，在对象字面量中有一种简洁生成器形式：

```
var a = {
    *foo() { .. }
};
```

我要说的是，有了简洁生成器，`*foo() { .. }` 比 `* foo() { .. }` 更自然。所以更进一步支持了与 `*foo()` 的一致性。

一致性易于理解和学习。

1. 运行生成器

尽管生成器用 * 声明，但执行起来还和普通函数一样：

```
foo();
```

你也可以传递参数给它，就像：

```
function *foo(x,y) {
    // ..
}

foo( 5, 10 );
```

主要的区别是，执行生成器，比如 foo(5,10)，并不实际在生成器中运行代码。相反，它会产生一个迭代器控制这个生成器执行其代码。

我们会在 3.3.2 节回到这个主题，现在简单地说就是：

```
function *foo() {
    // ..
}

var it = foo();

// 要启动/继续*foo()，调用it.next(..)
```

2. yield

生成器还有一个可以在其中使用的新关键字，用来标示暂停点：yield。考虑：

```
function *foo() {
    var x = 10;
    var y = 20;

    yield;

    var z = x + y;
}
```

在这个 *foo() 生成器中，首先执行前两行操作，然后 yield 会暂停这个生成器。如果恢复的话，恢复时会运行 *foo() 的最后一行。生成器中 yield 可以出现任意多次（严格说，或者根本不出现！）。

你甚至可以把 yield 放在循环中，用来表示一个重复暂停点。实际上，一个永不结束的循环就意味着一个永不结束的生成器，这是完全有效的，有时候完全就是你所需要的。

yield 不只是一个暂停点。它是一个表达式，在暂停生成器的时候发出一个值。这里是一个生成器中的 while..true 循环，每次迭代都会 yield 出一个新的随机数：

```
function *foo() {
    while (true) {
        yield Math.random();
    }
}
```

yield .. 表达式不只发送一个值——没有值的 yield 等价于 yield undefined——而且还会接收（也就是被替换为）最终的恢复值。考虑：

```
function *foo() {
    var x = yield 10;
    console.log( x );
}
```

这个生成器首先在暂停自身的时候 yield 出值 10。通过我们前面给出的 it.next(..) 恢复生成器的时候，恢复给定的值（如果有的话）就会替换 / 完成整个 yield 10 表达式，意味着这个值会被赋给变量 x。

yield.. 表达式可以出现在所有普通表达式可用的地方。举例来说：

```
function *foo() {
    var arr = [ yield 1, yield 2, yield 3 ];
    console.log( arr, yield 4 );
}
```

这里的 *foo() 有 4 个 yield.. 表达式。每一个 yield 都会导致这个生成器暂停等待一个恢复值，然后把这个恢复值用在各种表达式上下文中。

yield 严格上说不是一个运算符，尽管像 yield 1 这样使用它的时候确实看起来像是运算符。因为 yield 可以单独使用，比如 var x = yield;，把它当作运算符有时会令人迷惑。

严格来说，yield.. 和像 a = 3 这样的赋值表达式有同样的“表达式优先级”——类似于运算符优先级的概念。这意味着 yield.. 基本上可以出现在任何 a = 3 合法出现的位置。

让我们来考虑对称的这个例子：

```
var a, b;

a = 3;                  // 合法
b = 2 + a = 3;          // 不合法
b = 2 + (a = 3);        // 合法

yield 3;                // 合法
a = 2 + yield 3;        // 不合法
a = 2 + (yield 3);      // 合法
```

认真思考一下可以理解，yield.. 表达式和赋值表达式行为上的类似性有一定概念上的合理性。当一个暂停的 yield 表达式恢复的时候，它会被完成 / 替代为它的恢复值，采取的方式和“赋值”给这个值是一样的。

要点：如果需要 yield.. 出现在某个位置，而这个位置上像 a = 3 这样的赋值不允许出现，那么就要用 ( ) 封装。

因为 yield 关键字的优先级很低，几乎 yield.. 之后的任何表达式都会首先计算，然后再通过 yield 发送。只有 spread 运算符 ... 和逗号运算符 , 拥有更低的优先级，也就是说它

们会在 yield 已经被求值之后才会被绑定。

所以和普通语句中的多运算符一样，另外一个可能需要 ( ) 的情况是要覆盖（提升）yield 的低优先级，就像以下这些表达式的区别一样：

```
yield 2 + 3;            // 等价于yield (2 + 3)

(yield 2) + 3;          // 首先yield 2，然后+ 3
```

和 = 赋值一样，yield 也是"右结合"的，也就是说多个 yield 表达式连续出现等价于用 (..) 从右向左分组。所以，yield yield yield 3 会被当作 yield(yield(yield 3))。像 ((yield) yield) yield 3 这样的"左结合"解释是无意义的。

像对运算符一样，如果 yield 与其他运算符或者多个 yield 一起使用，通过 (..) 分组来澄清意图是好习惯，即使是在并不严格需要的时候。

> 要想获取其他关于运算符优先级和结合性的信息，参见本系列《你不知道的 JavaScript（中卷）》第一部分。

3. yield*

* 使得一个 function 声明成了 function* 生成器声明，类似地，* 使得 yield 成为了 yield *，这是一个完全不同的机制，称为 **yield 委托**（yield delegation）。语法上说，yield *.. 行为方式与 yield.. 完全相同，和我们上一小节讨论的一样。

yield * .. 需要一个 iterable；然后它会调用这个 iterable 的迭代器，把自己的生成器控制委托给这个迭代器，直到其耗尽。考虑：

```
function *foo() {
    yield *[1,2,3];
}
```

> 和生成器声明时的 * 位置一样（前面讨论过），* 的位置在 yield * 表达式中只是一个风格问题，可以由你自由选择。多数其他文献采用 yield* ..，而我喜欢 yield *..，原因和前面讨论过的类似。

值 [1,2,3] 产生了一个迭代器，一步输出一个值，所以 *foo() 生成器会随着消耗这些值把它们 yield 出来。展示这一特性的另一个方法是展示 yield 委托到另一个生成器：

```
function *foo() {
    yield 1;
    yield 2;
    yield 3;
}
```

```
function *bar() {
    yield *foo();
}
```

*bar() 调用 *foo() 的时候产生的迭代器通过 yield * 委托，这意味着不管 *foo() 产生什么值，这些值都会被 *bar() 产出。

使用 yield..，表达式的完成值来自于用 it.next(..) 恢复生成器的值，而对于 yield *.. 表达式来说，完成值来自于被委托的迭代器的返回值（如果有的话）。

正如我们在 3.1.4 节讨论过的，内置迭代器通常没有返回值。而如果自定义迭代器（或者生成器）的话，可以设计为 return 一个值，yield *.. 可以捕获这个值：

```
function *foo() {
    yield 1;
    yield 2;
    yield 3;
    return 4;
}

function *bar() {
    var x = yield *foo();
    console.log( "x:", x );
}

for (var v of bar()) {
    console.log( v );
}
// 1 2 3
// x: 4
```

值 1、2 和 3 从 *foo() 中 yield 出来后再从 *bar() 中 yield 出来，然后从 *foo() 返回的值 4 是 yield *foo() 表达式的完成值，被赋给了 x。

因为 yield * 可以调用另外一个生成器（通过委托到其迭代器），所以它也可以通过调用自身执行某种生成器递归：

```
function *foo(x) {
    if (x < 3) {
        x = yield *foo( x + 1 );
    }
    return x * 2;
}

foo( 1 );
```

foo(1) 以及之后的调用迭代器的 next() 来运行递归步骤的结果是 24。第一个 *foo(..) 运行 x 值为 1，满足 x < 3。x + 1 被递归地传给 *foo(..)，所以这一次 x 为 2。再次的递归调用使得 x 值为 3。

现在，因为不满足 x < 3，递归停止，返回 3 * 2 也就是 6 给前一个调用的 yield *.. 表达式，这个值被赋给 x。再次返回 6 * 2 的结果 12 给前一次调用的 x。最后是 12 * 2，也就是 24，返回给 *foo() 生成器的完成结果。

3.2.2 迭代器控制

前面我们简单介绍过生成器由迭代器控制这个概念。这里再深入探讨一下。

回忆一下前一小节中的递归 *foo(..)。下面是运行它的方式：

```
function *foo(x) {
    if (x < 3) {
        x = yield *foo( x + 1 );
    }
    return x * 2;
}

var it = foo( 1 );
it.next();              // { value: 24, done: true }
```

在上面的例子中，生成器没有真正暂停，因为并没有 yield .. 表达式。相反，yield * 只是通过递归调用保存当前的迭代步骤。所以，只要一次调用迭代器的 next() 函数就运行了整个生成器。

现在，让我们来考虑一个有多个步骤，因此有多个产生值的生成器：

```
function *foo() {
    yield 1;
    yield 2;
    yield 3;
}
```

我们已经知道，可以通过 for..of 循环消耗迭代器，即使是一个附着在 *foo() 这样的生成器上的迭代器：

```
for (var v of foo()) {
    console.log( v );
}
// 1 2 3
```

> for..of 循环需要一个 iterable。生成器函数引用（比如 foo）自己并不是一个 iterable；需要通过 foo() 执行它才能得到一个迭代器（也是一个 iterable，本章前面我们已经解释过）。理论上说可以为 GeneratorPrototype（所有生成器函数的原型）扩展一个主要就是 return this() 的 Symbol.iterator 函数。这会使得 foo 引用本身成为一个 iterable，也就是说 for (var v of foo) { .. }（注意 foo 上没有 ()）可以工作。

下面让我们来手动迭代这个生成器：

```
function *foo() {
    yield 1;
    yield 2;
    yield 3;
}

var it = foo();

it.next();              // { value: 1, done: false }
it.next();              // { value: 2, done: false }
it.next();              // { value: 3, done: false }

it.next();              // { value: undefined, done: true }
```

如果仔细观察可以看到，其中有 3 个 yield 语句和 4 个 next() 调用。这个不匹配看起来很奇怪。实际上，假定所有都被计算，生成器完整运行到结束，next() 调用总是会比 yield 语句多 1 个。

但是如果从相反的角度观察（由内向外而不是由外向内），yield 和 next() 的匹配更合理一些。

别忘了 yield.. 表达式用恢复生成器所用的值完成。这意味着传给 next(..) 的参数完成了当前 yield.. 表达式暂停等待完成的。

我们用以下方式说明这种思路：

```
function *foo() {
    var x = yield 1;
    var y = yield 2;
    var z = yield 3;
    console.log( x, y, z );
}
```

在这段代码中，每个 yield.. 从（1，2，3）中发出一个值，更直接地说，它是暂停生成器来等待一个值。换句话说几乎等价于在问“这里我应该用什么值？请回复。”这个问题。

下面是我们如何控制 *foo() 来启动它：

```
var it = foo();

it.next();              // { value: 1, done: false }
```

第一个 next() 调用初始的暂停状态启动生成器，运行直到第一个 yield。在调用第一个 next() 的时候，并没有 yield.. 表达式等待完成。如果向第一个 next() 调用传入一个值，这个值会马上被丢弃，因为并没有 yield 等待接收这个值。

“ES6 之后”的一个早期提案会通过生成器内部一个独立的元属性（参考第 7 章），支持访问传入最初 next(..) 调用的值。

现在，让我们来回答当前这个遗留问题，即“赋给 x 的值应该是什么？”我们通过发送一个值给**下一个** next(..) 调用来回答这个问题：

```
it.next( "foo" );           // { value: 2, done: false }
```

现在，x 的值就是 "foo"，但我们又提出了一个新问题，即“我们要给 y 赋什么值？”答案是：

```
it.next( "bar" );           // { value: 3, done: false }
```

给出答案，并提出一个新问题。最后答案是：

```
it.next( "baz" );          // "foo" "bar" "baz"
                           // { value: undefined, done: true }
```

现在应该更清楚每个 yield.. 的“问题”是如何由下一个 next(..) 调用来回答了，所以我们看到的“额外的”next() 调用就是启动所有这一切的第一个。

让我们把所有步骤集合到一起：

```
var it = foo();

// 启动生成器
it.next();                // { value: 1, done: false }

// 回答第一个问题
it.next( "foo" );         // { value: 2, done: false }

// 回答第二个问题
it.next( "bar" );         // { value: 3, done: false }

// 回答第三个问题
it.next( "baz" );         // "foo" "bar" "baz"
                          // { value: undefined, done: true }
```

你可以把生成器看作是值的产生器，其中每次迭代就是产生一个值来消费。

但是，从更通用的意义上来说，可能更合理的角度是把生成器看作一个受控的、可传递的代码执行，更像是 3.1.5 节中的 tasks 队列示例。

这个角度就是我们将在第 4 章中再次讨论生成器的动机。具体来说，并不需要 next(..) 在前一个 next(..) 结束后再被调用。在生成器的内部执行上下文被暂停时，程序的其余部分仍是未被阻塞的，包括控制生成器恢复的异步动作能力。

3.2.3 提前完成

本章前面讨论过，生成器上附着的迭代器支持可选的 `return(..)` 和 `throw(..)` 方法。这两种方法都有立即终止一个暂停的生成器的效果。

考虑：

```
function *foo() {
    yield 1;
    yield 2;
    yield 3;
}

var it = foo();

it.next();              // { value: 1, done: false }

it.return( 42 );        // { value: 42, done: true }

it.next();              // { value: undefined, done: true }
```

`return(x)` 有点像强制立即执行一个 `return x`，这样就能够立即得到指定值。一旦生成器完成，或者正常完毕或者像前面展示的那样提前结束，都不会再执行任何代码也不会返回任何值。

`return(..)` 除了可以手动调用，还可以在每次迭代的末尾被任何消耗迭代器的 ES6 构件自动调用，比如 `for..of` 循环和 spread 运算符 `...`。

这个功能的目的是通知生成器如果控制代码不再在它上面迭代，那么它可能就会执行清理任务（释放资源、重置状态等）。和普通的函数清理模式相同，完成这一点的主要方式是通过 `finally` 子句：

```
function *foo() {
    try {
        yield 1;
        yield 2;
        yield 3;
    }
    finally {
        console.log( "cleanup!" );
    }
}

for (var v of foo()) {
    console.log( v );
}
// 1 2 3
// cleanup!

var it = foo();
```

```
it.next();           // { value: 1, done: false }
it.return( 42 );     // cleanup!
                     // { value: 42, done: true }
```

不要把 yield 语句放在 finally 子句内部！虽然这是有效且合法的，但确实是一个可怕的思路。它会延后你的 return(..) 调用的完成，因为任何在 finally 子句内部的 yield.. 表达式都会被当作是暂停并发送消息；你不会像期望的那样立即得到完成的生成器。基本上不会有合理的原因要实现这么可怕的思路，所以不要这么用！

除了前面代码片段中展示的如何通过 return(..) 终止生成器，同时触发 finally 子句，它还展示了生成器在每次被调用的时候都产生了一个全新的迭代器。实际上，可以同时把多个迭代器附着在同一个生成器上：

```
function *foo() {
    yield 1;
    yield 2;
    yield 3;
}

var it1 = foo();
it1.next();             // { value: 1, done: false }
it1.next();             // { value: 2, done: false }

var it2 = foo();
it2.next();             // { value: 1, done: false }

it1.next();             // { value: 3, done: false }

it2.next();             // { value: 2, done: false }
it2.next();             // { value: 3, done: false }

it2.next();             // { value: undefined, done: true }
it1.next();             // { value: undefined, done: true }
```

提前终止

除了调用 return(..)，还可以调用 throw(..)。正如 return(x) 基本上就是在生成器中的当前暂停点插入了一个 return x，调用 throw(x) 基本上就相当于在暂停点插入一个 throw x。

除了对异常处理的不同（我们将在下一小节介绍对于 try 语句这意味着什么），throw(..) 同样引起提前完成，在当前暂停点终止生成器的运行。举例来说：

```
function *foo() {
    yield 1;
    yield 2;
    yield 3;
}
```

```
var it = foo();

it.next();                  // { value: 1, done: false }

try {
    it.throw( "Oops!" );
}
catch (err) {
    console.log( err );     // Exception: Oops!
}

it.next();                  // { value: undefined, done: true }
```

因为 throw(..) 基本上就是在生成器 yield 1 这一行插入一个 throw ..，没有处理这个异常，所以它会立即传递回调用代码，其中通过 try..catch 处理了这个异常。

和 return(..) 不同，迭代器的 throw(..) 方法从来不会被自动调用。

当然，尽管没有在前面代码中展示，如果在调用 throw(..) 的时候有 try..finally 子句在生成器内部等待，那么在异常传回调用代码之前 finally 子句会有机会运行。

3.2.4 错误处理

前面我们已经暗示，生成器的错误处理可以表达为 try..catch，它可以在由内向外和由外向内两个方向工作：

```
function *foo() {
    try {
        yield 1;
    }
    catch (err) {
        console.log( err );
    }

    yield 2;

    throw "Hello!";
}

var it = foo();

it.next();                 // { value: 1, done: false }

try {
    it.throw( "Hi!" );     // Hi!
                           // { value: 2, done: false }
    it.next();

    console.log( "never gets here" );
}
```

```
catch (err) {
    console.log( err ); // Hello!
}
```

错误也可以通过 `yield *` 委托在两个方向上传播：

```
function *foo() {
    try {
        yield 1;
    }
    catch (err) {
        console.log( err );
    }

    yield 2;

    throw "foo: e2";
}

function *bar() {
    try {
        yield *foo();

        console.log( "never gets here" );
    }
    catch (err) {
        console.log( err );
    }
}

var it = bar();

try {
    it.next();              // { value: 1, done: false }
    it.throw( "e1" );       // e1
                            // { value: 2, done: false }

    it.next();              // foo: e2
                            // { value: undefined, done: true }
}
catch (err) {
    console.log( "never gets here" );
}

it.next();                  // { value: undefined, done: true }
```

就像前面我们所看到的，`*foo()` 调用 `yield 1` 的时候，值 1 通过 `*bar()` 传递没有改变。

但这段代码最有趣的是，当 `*foo()` 调用 `throw "foo: e2"` 的时候，这个错误传播到了 `*bar()` 并立即被 `*bar()` 的 `try..catch` 代码块捕获。这个错误不会像值 1 一样穿过 `*bar()`。

接着 `*bar()` 的 `catch` 执行一个普通的输出 `err("foo: e2")` 然后 `*bar()` 正常执行完毕，这也是为什么从 `it.next()` 返回迭代器结果 `{ value: undefined, done: true }`。

如果 *bar() 在 yield *.. 表达式外并没有包裹一个 try..catch，那么这个错误当然就会一路传播出来，在路上还是会完成（终止）*bar() 的执行。

3.2.5 Transpile 生成器

可以用 ES6 之前的代码表达生成器功能吗？答案是可以，并且已经有好几个很棒的工具实现了这一点，包括最著名的 Facebook 的 Regenerator 工具（https://facebook.github.io/regenerator/）。

但为了更好地理解生成器，我们还是试着来手动转换一下。总的说来，我们将要创造一个简单的基于闭包的状态机。

我们的源生成器特别简单：

```
function *foo() {
    var x = yield 42;
    console.log( x );
}
```

一开始，我们需要一个名为 foo() 的函数来执行，它需要返回一个迭代器：

```
function foo() {
    // ..

    return {
        next: function(v) {
            // ..
        }

        // 省略return(..)和throw(..)
    };
}
```

现在，需要一些内部变量来记录在“生成器”逻辑步骤内部的当前位置。我们称之为 state。将会有 3 个状态：0 初始态、1 等待 yield 表达式完成、2 生成器完毕。

每次调用 next(..)，我们需要处理下一个步骤，然后增加 state。为了方便起见，我们把每个步骤放在一个 switch 语句的 case 子句中，并把这些放在内部函数 nextState() 中，next() 可以调用这个函数。另外，因为 x 是一个跨这个“生成器”整个作用域的变量，所以它需要存活在 nextState(..) 函数之外。

下面是所有代码（显然是某种程度上的简化，以便保持概念展示的清晰）：

```
function foo() {
    function nextState(v) {
        switch (state) {
            case 0:
```

```
                state++;

                // yield表达式
                return 42;
            case 1:
                state++;

                // yield表达式完成
                x = v;
                console.log( x );

                // 隐式return
                return undefined;

            // 不需要处理状态2
        }
    }

    var state = 0, x;

    return {
        next: function(v) {
            var ret = nextState( v );

            return { value: ret, done: (state == 2) };
        }

        // 省略return(..)和throw(..)
    };
}
```

最后，让我们来测试一下我们的前 ES6“生成器”：

```
var it = foo();

it.next();              // { value: 42, done: false }

it.next( 10 );          // 10
                        // { value: undefined, done: true }
```

还不错吧？希望这个练习帮你巩固了生成器实际上就是状态机逻辑的简化语法这个概念。这使得它们应用广泛。

3.2.6 生成器使用

现在，我们已经更深入理解了生成器的工作原理，那么它们适用于哪些场景呢？

我们已经看到了两个主要模式。

- *产生一系列值*

  这个用法可以很简单（比如随机字符串或者递增数），也可以表示更结构化的数据访问

（比如在数据库查询返回的行上迭代）。

不管怎样，我们使用迭代器来控制生成器，所以可以在每次调用 `next(..)` 的时候触发某些逻辑。数据结构上的普通迭代器只是取出值而没有控制逻辑。

- *顺序执行的任务队列*

  这种用法通常表示算法中步骤的流控制，其中每个步骤要求从某个外部源获得数据。每部分数据的完成可以是即时的，也可以是异步延迟的。

  从生成器内部代码的角度来看，在 `yield` 点同步或异步这样的细节是完全透明的。而且，这些细节是故意被抽象出去的，这样就不会被诸如实现复杂性模糊了步骤的自然顺序表达。抽象也意味着实现可以在无需修改生成器内部代码的情况下被替换 / 重构。

通过这些应用场景来观察生成器时，它们就远不止是手动状态机的不同或者说更优雅的语法形式了。它们是用于控制组织数据有序产生和消耗的强有力工具。

3.3 模块

在所有 JavaScript 代码中，唯一最重要的代码组织模式是模块，而且一直都是，我并不认为这是夸大其词。对于我本人，我认为也对于广泛社区来说，模块模式驱动了大多数代码。

3.3.1 旧方法

传统的模块模式基于一个带有内部变量和函数的外层函数，以及一个被返回的“public API”，这个“public API”带有对内部数据和功能拥有闭包的方法。通常这样表达：

```
function Hello(name) {
    function greeting() {
        console.log( "Hello " + name + "!" );
    }

    // public API
    return {
        greeting: greeting
    };
}

var me = Hello( "Kyle" );
me.greeting();          // Hello Kyle!
```

继续调用 `Hello(..)` 模块可以产生多个实例。有时一个模块只作为单例（singleton，也就是说只需要一个实例），这种情况下前面的代码需要稍作修改，通常这样使用一个 IIFE：

```
var me = (function Hello(name){
    function greeting() {
        console.log( "Hello " + name + "!" );
```

```
    }

    // public API
    return {
        greeting: greeting
    };
})( "Kyle" );

me.greeting();          // Hello Kyle!
```

这个模式是经过试验的。它也足够灵活，针对不同场景有多个变体。

其中常用的是异步模块定义（Asynchronous Module Definition，AMD），还有一种是通用模块定义（Universal Module Definition，UMD）。这里我们不会具体介绍这些模式和技术，但网上有很多详尽的解释。

3.3.2 前进

对于 ES6 来说，我们不再需要依赖于封装函数和闭包提供模块支持。ES6 中模块已经具备一等（first class）语法和功能支持。

在讨论具体语法细节之前，有一点很重要，就是要理解 ES6 模块和过去我们处理模块的方式之间的显著概念区别。

- ES6 使用基于文件的模块，也就是说一个文件一个模块。目前，还没有把多个模块合并到单个文件中的标准方法。

  这意味着如果想要把 ES6 模块直接加载到浏览器 Web 应用中，需要分别加载，而不是作为一大组放在单个文件中加载。在过去，为了性能优化，后者这种加载方式是很常见的。

  期待 HTTP/2 的到来能够显著消除所有这样的性能担忧，因为它运行在持久 socket 连接上，所以能够高效并发、交替加载多个小文件。
- ES6 模块的 API 是静态的。也就是说，需要在模块的公开 API 中静态定义所有最高层导出，之后无法补充。

  某些应用已经习惯了提供动态 API 定义的能力，可以根据对运行时情况的响应增加 / 删除 / 替换方法。这些用法或者改变自身以适应 ES6 静态 API，或者需要限制对二级对象属性 / 方法的动态修改。
- ES6 模块是单例。也就是说，模块只有一个实例，其中维护了它的状态。每次向其他模块导入这个模块的时候，得到的是对单个中心实例的引用。如果需要产生多个模块实例，那么你的模块需要提供某种工厂方法来实现这一点。
- 模块的公开 API 中暴露的属性和方法并不仅仅是普通的值或引用的赋值。它们是到内部模块定义中的标识符的实际绑定（几乎类似于指针）。

  在前 ES6 的模块中，如果把一个持有像数字或者字符串这样的原生值的属性放在公开 API 中，这个属性赋值是通过值复制赋值，任何对于对应变量的内部更新将会是独立

的，不会影响 API 对象的公开复制。

对于 ES6 来说，导出一个局部私有变量，即使当前它持有一个原生字符串 / 数字等，导出的都是到这个变量的绑定。如果模块修改了这个变量的值，外部导入绑定现在会决议到新的值。

- 导入模块和静态请求加载（如果还没加载的话）这个模块是一样的。如果是在浏览器环境中，这意味着通过网络阻塞加载；如果是在服务器上（比如 Node.js），则是从文件系统的阻塞加载。

  但是，不要惊慌于这里的性能暗示。因为 ES6 模块具有静态定义，导入需求可以静态扫描预先加载，甚至是在使用这个模块之前。

  关于如何处理这些加载请求，ES6 并没有实际指定或处理具体机制。这里有一个独立的模块加载器（Module Loader）的概念，其中每个宿主环境（浏览器、Node.js 等）提供一个适合环境的默认加载器。导入模块时使用一个字符串值表示去哪里获得这个模块（URL、文件路径等），但是这个值对于你的程序来说是透明的，只对加载器本身有意义。

  如果需要提供比默认加载器更细粒度的控制能力可以自定义加载器，默认加载器基本上没有粒度控制，因为它对于你的程序代码完全是不可见的。

你可以看到，ES6 模块将会为代码组织提供完整支持，包括封装、控制公开 API 以及引用依赖导入。但是实现方式非常特殊，可能可以也可能无法完全适应之前多年以来实现模块的方式。

CommonJS

还有一个类似、但并不完全兼容的模块语法 CommonJS，Node.js 生态系统下的开发者对此会很熟悉。

不得不说，长远看来，ES6 模块从本质上说必然会取代之前所有的模块格式和标准，即使是 CommonJS，因为 ES6 模块是建立在语言的语法支持基础上的。最终它总会不可逆转地作为统治方法成为最后的赢家。

但离那时还有很长的路要走。在服务器端 JavaScript 的世界里，已经有了成百上千的 CommonJS 风格模块，以及浏览器端十倍于此的各种格式标准的模块（UMD、AMD、临时性的模块方案）。要迁移这些模块需要很多年才能初见成效。

在此期间，模块 transpiler / 转换工具是必不可少的。你可能刚刚开始习惯这个新现实。不管你是在编写普通模块、AMD、UMD、CommonJS 还是 ES6，这些工具都不得不解析转化为对代码运行的所有环境都适用的形式。

对于 Node.js 来说，这可能意味着（目前的）目标是 CommonJS。对于浏览器来说，可能是 UMD 或者 AMD。接下来的几年里，随着这些工具的成熟和最佳实践的涌现，这一方面可能会变幻莫测。

由此，关于模块我的最好建议是：不管之前你虔诚地支持和熟悉哪种格式，现在开始开发和理解 ES6 模块吧，因为它终将会使得其他模块方法消失。它是 JavaScript 模块的未来，虽然现在有点落后。

3.3.3　新方法

支撑 ES6 模块的两个主要新关键字是 `import` 和 `export`。它们在语法上有许多微妙之处，所以我们来深入了解一下。

> 这里有一个很容易被忽略的重要细节：`import` 和 `export` 都必须出现在使用它们的最顶层作用域。举例来说，不能把 `import` 或 `export` 放在 `if` 条件中；它们必须出现在所有代码块和函数的外面。

1. 导出 API 成员

`export` 关键字或者是放在声明的前面，或者是作为一个操作符（或类似的）与一个要导出的绑定列表一起使用。考虑：

```
export function foo() {
    // ..
}

export var awesome = 42;

var bar = [1,2,3];
export { bar };
```

下面是同样导出的另外一种表达形式：

```
function foo() {
    // ..
}

var awesome = 42;
var bar = [1,2,3];

export { foo, awesome, bar };
```

这些都称为**命名导出**（named export），因为导出变量 / 函数等的名称绑定。

没有用 `export` **标示**的一切都在模块作用域内部保持私有。也就是说，尽管 `var bar = ..` 看起来像是声明在全局作用域的顶层，而这个顶层作用域实际上是模块本身；在模块内没有全局作用域。

> 模块还能访问 `window` 和所有的“全局”变量，只是不作为词法上的顶层作用域。但如果可能的话，在你的模块里应该尽量远离那些全局量。

在命名导出时还可以“重命名”（也即别名）一个模块成员：

```
function foo() { .. }

export { foo as bar };
```

导入这个模块的时候，只有成员名称 bar 可以导入；foo 还是隐藏在模块内部。

模块导出不是像你熟悉的赋值运算符 = 那样只是值或者引用的普通赋值。实际上，导出的是对这些东西（变量等）的绑定（类似于指针）。

如果在你的模块内部修改已经导出绑定的变量的值，即使是已经导入的（参见下一小节），导入的绑定也将会决议到当前（更新后）的值。

考虑：

```
var awesome = 42;
export { awesome };

// 之后
awesome = 100;
```

导入这个模块的时候，不管是在 awesome = 100 之前还是之后，一旦赋值发生，导入的绑定就会决议到 100 而不是 42。

这是因为本质上，绑定是一个指向 awesome 变量本身的引用或者指针，而不是这个值的复制。ES6 模块绑定为 JavaScript 带来的这个概念是前所未有的。

尽管显然可以在模块定义内部多次使用 export，ES6 绝对倾向于一个模块使用一个 export，称之为**默认导出**（default export）。TC39 委员会的一些成员认为，如果遵循这个模式，那么“得到的回报是更简单的 import 语法”，如果不这么做，得到的“惩罚”则是更繁复的语法。

默认导出把一个特定导出绑定设置为导入模块时的默认导出。绑定的名称就是 default。后面将会看到，导入模块绑定时可以重命名，因为通常都会使用默认导出。

每个模块定义只能有一个 default。下一小节将会介绍 import，那时你可以看到如果模块有一个默认导出，它将如何使得 import 语法更加简洁。

关于默认导出有一个微妙的细节需要格外小心。比较这两段代码：

```
function foo(..) {
    // ..
}

export default foo;
```

以及

```
function foo(..) {
    // ..
}

export { foo as default };
```

在第一段代码中，导出的是此时到函数表达式值的绑定，而不是标识符 foo。换句话说，export default .. 接受的是一个表达式。如果之后在你的模块中给 foo 赋一个不同的值，模块导入得到的仍然是原来导出的函数，而不是新的值。

顺便说一下，第一段代码也可以这么写：

```
export default function foo(..) {
    // ..
}
```

> 尽管这里的 function foo.. 部分严格说是一个函数表达式，但由于模块内部作用域的原因，它被当作函数声明对待，因为 foo 名称和模块的最高级作用域绑定（通常称为“hoisting”）。export default class Foo.. 也是这样。然而，虽然你可以 export var foo = ..，但是现在还不能 export default var foo = ..（或者 let 或者 const），这种不一致很令人烦恼。在编写本部分时，已经有讨论建议为了一致性尽快在后 ES6 时期增加这个功能。

再次回忆一下第二段代码：

```
function foo(..) {
    // ..
}

export { foo as default };
```

在这个版本的模块导出中，默认导出绑定实际上绑定到 foo 标识符而不是它的值，所以得到了前面描述的绑定行为（也就是说，如果之后修改了 foo 的值，在导入一侧看到的值也会更新）。

要十分小心默认导出语法的这个微妙陷阱，特别是在代码逻辑需要更新导出值的时候。如果你并不打算更新默认导出的值，那么使用 export default .. 就好。如果确实需要更新这个值，就需要使用 export { .. as default }。不管怎样，记得通过代码注释来解释你的意图！

因为每个模块只能有一个 default，所以你可能会忍不住把模块设计成默认导出一个对象，其中包含所有的 API 方法，比如：

```
export default {
    foo() { .. },
    bar() { .. },
    ..
};
```

这个模式看起来与大量开发者已经构造的前 ES6 模块紧密呼应，所以似乎是一个自然而然的方法。但不幸的是，它有一些缺陷，同时也是官方不建议采用的。

具体来说，JavaScript 引擎无法静态分析平凡对象的内容，这意味着它无法对静态 `import` 进行性能优化。让每个成员独立且显式地导出的优点是引擎可以对其进行静态分析和优化。

如果你的 API 已有多个成员，这些原则——每个模块一个默认导出，所有的 API 成员作为命名导出——看起来似乎相互冲突，不是吗？但你可以有一个单独的默认导出，同时又有其他命名导出；它们并不互斥。

所以，要取代这个（不推荐的）模式：

```
export default function foo() { .. }

foo.bar = function() { .. };
foo.baz = function() { .. };
```

可以用

```
export default function foo() { .. }

export function bar() { .. }
export function baz() { .. }
```

> 在前面的代码中，我使用名称 `foo` 作为 `default` 标识的函数。然而，这个名称 `foo` 对于导出来说是忽略的——实际上导出的名字是 `default`。下一小节将会介绍，在导入这个默认绑定的时候，可以任意地给它起一个名称。

也有人更喜欢这种形式：

```
function foo() { .. }
function bar() { .. }
function baz() { .. }

export { foo as default, bar, baz, .. };
```

后面很快会介绍 `import`，到那时混用默认和命名导出的效果将会更加清晰。但从本质上说，这意味着最简单的默认导入形式只会提取函数 `foo()`。如果需要的话，用户可以继续手动列出 `bar` 和 `baz` 作为命名导入。

可以设想一下，如果提供大量命名导出绑定，那么对模块的用户来说将会是多么麻烦。有一个通配符导入可以用于把模块的所有导出导入到单个命名空间对象中，但无法通配符导入到顶层绑定。

再一次重申，ES6 模块机制的设计意图是不鼓励模块大量导出；相对而言，它是有意想让这样的方法麻烦一些，作为某种社会工程来提倡简单模块设计，而不是大型 / 复杂模块设计。

我可能会建议你避免混用默认导出和命名导出，特别是在有大量 API 并且通过重构拆分模块不实际或者不想这么做的时候。这种情况下，就都用命名导出好了，提供文档说明模块用户会用 `import * as ..`（名字空间导入，下一小节将会介绍）方法来一次把所有 API 引入到某个名字空间中。

前面已经介绍过，但这里让我们再详细讨论一下。除了 `export default ...` 形式导出一个表达式值绑定，所有其他的导出形式都是导出局部标识符的绑定。对于这些绑定来说，如果导出之后在模块内部修改某个值，外部导入的绑定会访问到修改后的值：

```
var foo = 42;
export { foo as default };

export var bar = "hello world";

foo = 10;
bar = "cool";
```

当你导入这个模块的时候，`default` 和 `bar` 导出会绑定到局部变量 `foo` 和 `bar`，也就是说它们会暴露更新后的值 `10` 和 `"cool"`。导出时刻的值无关紧要。导入时刻的值也无关紧要。绑定是活连接，所以重要的是访问这个绑定时刻的当前值。

> 双向绑定是不允许的。如果从一个模块导入了 `foo`，然后修改导入的 `foo` 变量的值，就会抛出错误！下一小节我们会再次介绍这一点。

你也可以再次导出某个模块的导出，就像这样：

```
export { foo, bar } from "baz";
export { foo as FOO, bar as BAR } from "baz";
export * from "baz";
```

这些形式类似于首先从 `"baz"` 模块导入，然后显式列出它的成员再从你的模块导出。但这些形式中，`"baz"` 模块的成员不会导入到你的模块的局部作用域，它们就像是不留痕迹地穿过。

2. 导入 API 成员

不出意料，使用 `import` 语句导入模块。就像 `export` 有几种变体，`import` 也是一样，所以

花点时间思考接下来的问题，并验证一下你的选择。

如果想导入一个模块 API 的某个特定命名成员到你的顶层作用域，可以使用下面语法：

```
import { foo, bar, baz } from "foo";
```

这里的 { .. } 语法可能看起来像是一个对象字面量，或者甚至是一个对象解构语法。但这种形式是专用于模块的，所以注意不要把它和其他 { .. } 模式混淆。

字符串 "foo" 称为**模块指定符**（module specifier）。因为整体目标是可静态分析的语法，模块指定符必须是字符串字面值，而不能是持有字符串值的变量。

从 ES6 代码和 JavaScript 引擎本身的角度来说，这个字符串字面量的内容是完全透明的，也毫无意义。模块加载器会把这个字符串解释为一个决定去哪儿寻找所需模块的指令，或者作为 URL 路径或者是本地文件系统路径。

列出的标识符 foo、bar 和 baz 必须匹配模块 API 的命名导出（会应用静态分析和错误判定）。它们会在当前作用域绑定为顶层标识符：

```
import { foo } from "foo";

foo();
```

你可以对导入绑定标识符重命名，就像这样：

```
import { foo as theFooFunc } from "foo";

theFooFunc();
```

如果这个模块只有一个你想要导入并绑定到一个标识符的默认导出，绑定时可以省略包围的 { .. } 语法。这种情况下的 import 得到了最简洁优美的 import 语法形式：

```
import foo from "foo";

// 或者：
import { default as foo } from "foo";
```

前面一小节介绍过，模块的 export 中的关键字 default 指定了一个命名导出，名称实际上就是 default，就像第二种更详细的语法形式表明的一样。在后一种语法中，从 default 到这个例子中的 foo 的重命名都是显式的，和前一种隐式语法形式是一样的。

你还可以把默认导出与其他命名导出一起导入，如果这个模块有这样的定义的话。回忆前

面这个模块定义：

```
export default function foo() { .. }

export function bar() { .. }
export function baz() { .. }
```

导入这个模块的默认导出和它的两个命名导出：

```
import FOOFN, { bar, baz as BAZ } from "foo";

FOOFN();
bar();
BAZ();
```

ES6 模块哲学强烈建议的方法是，只从模块导入需要的具体绑定。如果一个模块提供了 10 个 API 方法，但是你只需要其中的 2 个，有些人坚信把所有的 API 绑定都导入进来是一种浪费。

除了代码更清晰，窄导入的另一个好处是使得静态分析和错误检测（比如意外使用了错误的绑定名称）更加健壮。

当然，ES6 设计哲学只影响了标准立场，没有任何强制要求必须采用这种方法。

很多开发者很快会发现这种方法可能更繁复，因为每次意识到需要模块中的新东西的时候，都要重新访问和更新 `import` 语句。因此这种方法要权衡考虑的是便捷性。

考虑到这一点，理想的选择是从模块把所有一切导入到一个单独命名空间，而不是向作用域直接导入独立的成员。幸运的是，`import` 语句有一种语法变体可以支持这种模块导入，称为**命名空间导入**（namespace import）。

考虑一个模块 `"foo"` 导出，如下：

```
export function bar() { .. }
export var x = 42;
export function baz() { .. }
```

你可以把整个 API 导入到单个模块命名空间绑定：

```
import * as foo from "foo";

foo.bar();
foo.x;          // 42
foo.baz();
```

这个 `* as ..` 语句需要一个 * 通配符。换句话说，不能用 `import { bar, x } as foo from "foo"` 这样的语句只导入 API 的一部分但仍然绑定到 `foo` 命名空间。我希望有这样的支持，但是 ES6 命名空间导入是要么全有要么全无的。

如果通过 `* as ..` 导入的模块有默认导出，它在指定的命名空间中的名字就是 `default`。你还可以在这个命名空间绑定之外把默认导入作为顶层标识符命名。考虑一个模块 `"world"` 导出，如下：

```
export default function foo() { .. }
export function bar() { .. }
export function baz() { .. }
```

以及下面的导入：

```
import foofn, * as hello from "world";

foofn();
hello.default();
hello.bar();
hello.baz();
```

虽然这个语法是合法的，但可能会令人迷惑，因为这个模块的一个方法（默认导出）绑定到了作用域的顶层，而其余的命名导出（其中一个名为 `default`）绑定到了另一个（`hello`）标识符命名空间。

前面已经提到过，我建议避免这样设计模块导出，为的是尽量避免模块用户被这种奇怪的设计所迷惑。

所有导入的绑定都是不可变和 / 或只读的。考虑一下前面的导入，导入之后所有这些试图赋值的动作都会抛出 `TypeErrors`：

```
import foofn, * as hello from "world";

foofn = 42;          // （运行时）TypeError!
hello.default = 42;  // （运行时）TypeError!
hello.bar = 42;      // （运行时）TypeError!
hello.baz = 42;      // （运行时）TypeError!
```

回忆一下 3.3.3 节，其中讨论了 `bar` 和 `baz` 绑定是如何绑定到模块 `"world"` 内部的实际标识符上的。这意味着如果模块修改了这些值，`hello.bar` 和 `hello.baz` 现在指向修改后的新值。

但是你的局部导入绑定的不变性 / 只读性限制了无法从导入的绑定修改它们，否则就会 `TypeErrors`。这是非常重要的，因为如果没有这样的保护，你的修改就会最终影响这个模块的所有其他用户（别忘了：单例），这会导致出乎意料的副作用！

另外，尽管模块可以从内部修改 API 成员，但如果需要故意这样设计还是要格外小心。ES6 模块应该是静态的，所以要尽可能少地偏离这个原则，并且应该用文档加以认真详尽地说明。

有一些设计哲学实际上想让用户修改 API 的某个属性值，或者模块 API 被设计成是通过增加到 API 命名空间的“插件”来扩展的。前面我们断言，ES6 模块 API 应该被认为或者被设计成静态不可变的，这严格限制和抑制了这些另类的模块设计模式。你可以通过导出平凡对象来绕过这些限制，这个对象当然可以修改。但是要这么做之前一定要三思而后行。

作为 import 结果的声明是“提升的”（参见本系列《你不知道的 JavaScript（上卷）》第一部分）。考虑：

```
foo();

import { foo } from "foo";
```

foo() 可以运行，不只是因为 import .. 语句的静态决议在编译过程中确定了 foo 值是什么，也因为它“提升”了在模块作用域顶层的声明，使它在模块所有位置可用。

最后，import 最基本的形式是这样的：

```
import "foo";
```

这种形式并没有实际导入任何一个这个模块的绑定到你的作用域。它加载（如果还没有加载的话）、编译（如果还没有编译的话），并求值（如果还没有运行的话）"foo" 模块。

一般来说，这种导入没什么太大用处。可能有一些模块定义有副作用（比如把东西赋给 window / 全局对象）的情况。你也可以把 import "foo" 想象成是对以后可能需要的模块的预加载。

3.3.4　模块依赖环

A 导入 B，B 导入 A。这种情况到底是怎么工作的？

首先必需声明，我尽量避免故意设计带有环形依赖的系统。前面已经说过，我意识到有一些原因导致人们需要这么做，因为它可以解决某些棘手的设计情况。

让我们看一下 ES6 是怎么处理这个问题的。首先，模块 "A"：

```
import bar from "B";

export default function foo(x) {
    if (x > 10) return bar( x - 1 );
    return x * 2;
}
```

然后，模块 "B"：

```
import foo from "A";
```

```
export default function bar(y) {
    if (y > 5) return foo( y / 2 );
    return y * 3;
}
```

foo(..) 和 bar(..) 这两个函数如果在同一个作用域的话，将会作为标准函数声明，因为这两个声明是“提升”到整个作用域的，所以不管代码顺序如何都可以访问彼此。

有了模块，那么声明就是在完全不同的作用域，所以 ES6 需要额外的工作来支持这样的循环引用。

下面是从粗略概念的意义上循环的 import 依赖如何生效和解析的过程。

- 如果先加载模块 "A"，第一步是扫描这个文件分析所有的导出，这样就可以注册所有可以导入的绑定。然后处理 import .. from "B"，这表示它需要取得 "B"。
- 引擎加载 "B" 之后，会对它的导出绑定进行同样的分析。当看到 import .. from "A"，它已经了解 "A" 的 API，所以可以验证 import 是否有效。现在它了解 "B" 的 API，就可以验证等待的 "A" 模块中 import .. from "B" 的有效性。

本质上说，相互导入，加上检验两个 import 语句的有效性的静态验证，虚拟组合了两个独立的模块空间（通过绑定），这样 foo(..) 可以调用 bar(..)，反过来也是一样。这和如果它们本来是声明在同一个作用域中是对称的。

现在让我们试着来使用这两个模块。首先，试一下 foo(..)：

```
import foo from "foo";
foo( 25 );              // 11
```

也可以使用 bar(..)：

```
import bar from "bar";
bar( 25 );              // 11.5
```

在 foo(25) 或 bar(25) 调用执行的时候，所有模块的所有分析 / 编译都已经完成。这意味着 foo(..) 内部已经直接了解 bar(..)，而 bar(..) 内部也已经直接了解 foo(..)。

如果只是需要与 foo(..) 交互，那么只需要导入 "foo" 模块。对于 bar(..) 和 "bar" 模块也是一样。

当然，需要的话可以导入并使用二者：

```
import foo from "foo";
import bar from "bar";

foo( 25 );              // 11
bar( 25 );              // 11.5
```

import 语句的静态加载语义意味着可以确保通过 import 相互依赖的 "foo" 和 "bar" 在其中任何一个运行之前，二者都会被加载、解析和编译。所以它们的环依赖是静态决议的，就像期望的一样。

3.3.5 模块加载

我们在 3.3 节开始部分介绍过，import 语句使用外部环境（浏览器、Node.js 等）提供的独立机制，来实际把模块标识符字符串解析成可用的指令，用于寻找和加载所需的模块。这个机制就是系统**模块加载器**。

如果在浏览器中，环境提供的默认模块加载器会把模块标识符解析为 URL，（一般来说）如果在像 Node.js 这样的服务器上就解析为本地文件系统路径。默认行为方式假定加载的文件是以 ES6 标准模块格式编写的。

另外，还可以通过 HTML 标签加载模块到浏览器中，这与目前脚本程序加载的方式类似。编写本部分的时候，还不清楚这个标签是 <script type="module"> 还是 <module>。这并非由 ES6 决定，ES6 讨论的同时在合适的规范中对此已经有了很多讨论。

不管这个标签看起来是什么样，可以确定的是在底层将会使用默认加载器（或者是我们将在下一小节介绍的预先指定的自定义加载器）。

就像在 markup 中使用的标签一样，模块加载器本身不是由 ES6 指定的。它是独立的、平行的标准（http:// whatwg.github.io/loader/），目前由 WHATWG 浏览器标准工作组管理。

接下来的讨论反映了编写本部分时 API 设计上的一个初期版本，未来很可能改变。

1. 在模块之外加载模块

直接与模块加载器交互的一个用法是非模块需要加载一个模块的情况。考虑：

```
// 一般脚本在浏览器中通过<script>加载，这里import不合法

Reflect.Loader.import( "foo" ) // 为"foo"返回一个promise
.then( function(foo){
    foo.bar();
} );
```

工具 Reflect.Loader.import(..) 把整个模块导入到命名参数（作为一个命名空间），就像前面讨论的命名空间导入 import * as foo .. 一样。

> Reflect.Loader.import(..) 工具返回一个 promise，这个 promise 模块就绪就会完成。要导入多个模块，可以使用 Promise.all([..]) 组合多个 Reflect.Loader.import(..) 调用返回的 promise。关于 Promise 的更多信息，参见 4.1 节。

你也可以在真正的模块中使用 `Reflect.Loader.import(..)` 来动态 / 有条件地加载一个模块，而 `import` 本身无法实现。比如，你可能想要在测试表明当前引擎没有定义某个 ES7+ 特性的情况下才加载一个包含这个特性 polyfill 的模块。

出于性能的考虑，需要尽可能避免动态加载，因为这会妨碍 JavaScript 引擎根据静态分析提前获取代码的能力。

2. 自定义加载

另外一种与模块加载器直接交互的用法，就是需要通过配置甚至重定义来自定义其行为的情况。

在编写本部分的时候，已经有一个模块加载器 API 的 polyfill 在开发之中了（https://github.com/ModuleLoader/es6-module-loader）。因为详细信息还比较少，也很可能会变化，所以我们来探索一下几种最终可能出现的特性。

`Reflect.Loader.import(..)` 调用可能会支持第二个参数，用来指定自定义导入 / 加载任务的各种选项。比如：

```
Reflect.Loader.import( "foo", { address: "/path/to/foo.js" } )
.then( function(foo){
    // ..
})
```

还可能会（通过某些手段）提供自定义支持来嵌入加载模块的过程，在加载完成引擎编译模块之前可以执行一个转换 /transpilation。

举例来说，可以加载某些不符合 ES6 规范的模块格式（比如 CoffeeScript、TypeScript、CommonJS 和 AMD）。转换步骤把它转换为遵循 ES6 规范的模块，然后引擎处理。

3.4 类

几乎从 JavaScript 发展初期开始，语法和开发模式都极力营造支持面向对象开发的假象。通过诸如 `new` 和 `instanceof` 以及 `.constructor` 属性，让人们忍不住以为 JavaScript 在其原型系统内部某处隐藏着类。

当然，JavaScript“类”和传统的类并不相同。二者的区别已经有文档详尽说明，关于这点这里我不再赘述。

> 要进一步学习 JavaScript 中用来模拟“类”的模式，以及名为“委托”的这种对原型的另外一种看法，参见本系列《你不知道的 JavaScript（上卷）》第二部分的后半部分。

3.4.1 class

尽管 JavaScript 的原型机制并不像传统类那样工作，但这没有阻止要求这个语言扩展其语法糖使其能够更像真正的类那样表达“类”这种强烈的趋势。因此有了 ES6 `class` 关键字及其相关的机制。

这个特性是具有强烈争议性的讨论结果，代表着关于如何实现 JavaScript 类的几种强烈冲突观点的更小的妥协子集。多数想要 JavaScript 完整类支持的开发者会发现这个新语法的部分十分诱人，同时也会发现还有一些要点仍然缺失。但别着急，TC39 已经开始研究在 ES6 后提供更多的特性来支持类。

新的 ES6 类机制的核心是关键字 `class`，表示一个**块**，其内容定义了一个函数原型的成员。考虑：

```
class Foo {
    constructor(a,b) {
        this.x = a;
        this.y = b;
    }

    gimmeXY() {
        return this.x * this.y;
    }
}
```

需要注意以下几点。

- `class Foo` 表明创建一个（具体的）名为 `Foo` 的函数，与你在前 ES6 中所做的非常类似。
- `constructor(..)` 指定 `Foo(..)` 函数的签名以及函数体内容。
- 类方法使用第 2 章讨论过的对象字面量可用的同样的“简洁方法”语法。这也包含本章前面讨论过的简洁生成器形式，以及 ES5 getter/setter 语法。但是，类方法是不可枚举的，而对象方法默认是可枚举的。
- 和对象字面量不一样，在 `class` 定义体内部不用逗号分隔成员！实际上，这甚至是不允许的。

可以把前面代码中的 `class` 语法定义粗略理解为下面这个等价前 ES6 代码，之前有过原型风格编码经验的开发者可能会对此非常熟悉：

```
function Foo(a,b) {
    this.x = a;
    this.y = b;
}

Foo.prototype.gimmeXY = function() {
    return this.x * this.y;
}
```

不管是前 ES6 形式还是新的 ES6 class 形式，现在都可以按照期望来实例化和使用这个“类”：

```
var f = new Foo( 5, 15 );

f.x;                        // 5
f.y;                        // 15
f.gimmeXY();                // 75
```

注意！尽管 class Foo 看起来很像 function Foo()，但二者有重要区别。

- 由于前 ES6 可用的 Foo.call(obj) 不能工作，class Foo 的 Foo(..) 调用必须通过 new 来实现。
- function Foo 是“提升的”（参见本系列《你不知道的 JavaScript（上卷）》第一部分），而 class Foo 并不是；extends .. 语句指定了一个不能被“提升”的表达式。所以，在实例化一个 class 之前必须先声明它。
- 全局作用域中的 class Foo 创建了这个作用域的一个词法标识符 Foo，但是和 function Foo 不一样，并没有创建一个同名的全局对象属性。

因为 class 只是创建了一个同名的构造器函数，所以现有的 instanceof 运算符对 ES6 类仍然可以工作。然而，ES6 引入了一种使用 Symbol.hasInstance（参见 7.3 节）自定义 instanceof 如何工作的方法。

还有一种我认为更方便的思考类的方式，就是把它看作一个**宏**（macro），用于自动产生一个 prototype 对象。可选的是，如果使用 extends，也连接起了 [[Prototype]] 关系。

ES6 class 本身并不是一个真正的实体，而是一个包裹着其他像函数和属性这样的具体实体并把它们组合到一起的元概念。

> 除了声明形式，class 也可以是一个表达式，就像在这一句中：var x = class Y { .. }。对于把类定义（严格说，是构造器本身）作为函数参数传递，或者把它赋给一个对象属性的时候特别有用。

3.4.2 extends 和 super

ES6 类还通过面向类的常用术语 extends 提供了一个语法糖，用来在两个函数原型之间建立 [[Prototype]] 委托链接——通常被误称为“继承”或者令人迷惑地标识为“原型继承”：

```
class Bar extends Foo {
    constructor(a,b,c) {
        super( a, b );
        this.z = c;
```

```
    }

    gimmeXYZ() {
        return super.gimmeXY() * this.z;
    }
}

var b = new Bar( 5, 15, 25 );

b.x;                        // 5
b.y;                        // 15
b.z;                        // 25
b.gimmeXYZ();               // 1875
```

还有一个重要的新增特性是 super，这在前 ES6 中实际上是无法直接支持的（如果不使用某种不幸的 hack 权衡的话）。在构造器中，super 自动指向"父构造器"，在前面的例子中就是 Foo(..)。在方法中，super 会指向"父对象"，这样就可以访问其属性 / 方法了，比如 super.gimmeXY()。

Bar extends Foo 的意思当然就是把 Bar.prototype 的 [[Prototype]] 连接到 Foo.prototype。所以，在像 gimmeXYZ() 这样的方法中，super 具体指 Foo.prototype，而在 Bar 构造器中 super 指的是 Foo。

super 并不仅限于在 class 声明中使用。它还可以在对象字面值中使用，用法和我们这里介绍的几乎一样。参见 2.6.5 节获取更多信息。

1. super 恶龙

super 的行为根据其所处的位置不同而有所不同，这一点值得注意。公平地说，绝大多数情况下这不是一个问题。但如果你偏离了狭窄的正轨就会出现意外。

可能会存在这种情况，即你想要在构造器中引用 Foo.prototype，比如要直接访问它的某个属性或方法。但是，构造器中不能这样使用 super；super.protytype 不能工作。简单地说 super(..) 意味着调用 new Foo(..)，但是实际上并不是指向 Foo 自身的一个可用引用。

同样地，你可能想要在非构造器方法内部引用 Foo(..) 函数。super.constructor 指向函数 Foo(..)，但是请注意这个函数**只能**通过 new 调用。new super.constructor 是合法的，不过它在多数情况下没什么用处，因为你无法让这个调用使用或引用当前的 this 对象上下文，而这很可能才是你需要的。

还有，看起来似乎 super 像 this 一样可以被函数上下文驱动，也就是说，它们都能动态绑定。但是，super 并不像 this 那样是动态的。构造器或函数在声明时在内部建立了 super 引用（在 class 声明体内），此时 super 是静态绑定到这个特定的类层次上的，不能重载（至少在 ES6 中是这样）。

这意味着什么呢？它意味着如果你习惯于调用一个“类”的方法时通过覆盖它的 this 来从另外一个类中“借来”这个方法，比如通过 call() 或 apply(..)，那么如果借用的这个方法内有一个 super，结果很可能出乎意料。考虑下面这个类层次：

```
class ParentA {
    constructor() { this.id = "a"; }
    foo() { console.log( "ParentA:", this.id ); }
}

class ParentB {
    constructor() { this.id = "b"; }
    foo() { console.log( "ParentB:", this.id ); }
}

class ChildA extends ParentA {
    foo() {
        super.foo();
        console.log( "ChildA:", this.id );
    }
}

class ChildB extends ParentB {
    foo() {
        super.foo();
        console.log( "ChildB:", this.id );
    }
}

var a = new ChildA();
a.foo();                // ParentA: a
                        // ChildA: a
var b = new ChildB();   // ParentB: b
b.foo();                // ChildB: b
```

在前面的代码中，看起来一切似乎都自然而然。但如果你想要借用 b.foo() 并在 a 的上下文中使用它——通过动态 this 绑定，这样的借用是十分常见的，并且有多种不同的使用方法，包括最著名 mixins——你可能会发现结果不幸地出乎意料：

```
// 在a的上下文中借来b.foo()使用
b.foo.call( a );            // ParentB: a
                            // ChildB: a
```

可以看到，this.id 引用被动态重新绑定，因此两种情况下都打印： a，而不是： b。但是 b.foo() 的 super.foo() 引用没有被动态重绑定，所以它仍然打印出 ParentB 而不是期望的 ParentA。

因为 b.foo() 引用了 super，它是静态绑定到 ChildB/ParentB 类层次的，而不能用在 ChildA / ParentA 类层次。ES6 并没有对这个局限提供解决方案。

如果你有一个静态类层次，而没有“异花传粉”的话，super 似乎可以按照直觉期望

工作。但公正地说，实现 this 感知编码的主要好处之一就是这种灵活性。简单地说，class+super 要求避免使用这样的技术。

最后的选择归结为把对象设计限制在这些静态类层次之内——class、extends 和 super 就好——或者放弃“模拟”类的企图转而拥抱动态和灵活性，拥抱非类对象和 [[Prototype]] 委托（参见本系列《你不知道的 JavaScript（上卷）》第二部分）。

2. 子类构造器

对于类和子类来说，构造器并不是必须的；如果省略的话那么二者都会自动提供一个默认构造器。但是，这个默认替代构造器对于直接类和扩展类来说有所不同。

具体来说，默认子类构造器自动调用父类的构造器并传递所有参数。换句话说，可以把默认子类构造器看成下面这样：

```
constructor(...args) {
    super(...args);
}
```

这是一个需要注意的重要细节。并不是在所有支持类的语言中子类构造器都会自动调用父类构造器。C++ 会这样，Java 则不然。更重要的是，在前 ES6 的类中，并不存在这样的自动“父类构造器”调用。如果你的代码依赖的这样的调用**没有**发生，那么转换为 ES6 class 的时候要格外小心。

另外一个 ES6 子类构造器或许是出乎意料的偏离 / 限制是：子类构造器中调用 super(..) 之后才能访问 this。其原因比较微妙复杂，但可以归结为创建 / 初始化你的实例 this 的实际上是父构造器。前 ES6 中，它的实现正相反；this 对象是由“子类构造器”创建的，然后在子类的 this 上下文中调用“父类”构造器。

下面来说明一下。以下代码在前 ES6 中可以工作：

```
function Foo() {
    this.a = 1;
}

function Bar() {
    this.b = 2;
    Foo.call( this );
}

// `Bar` "extends" `Foo`
Bar.prototype = Object.create( Foo.prototype );
```

而这个等价 ES6 代码则不合法：

```
class Foo {
    constructor() { this.a = 1; }
```

```
}

class Bar extends Foo {
    constructor() {
        this.b = 2;     // 不允许在super()之前
        super();        // 要改正的话可以交换这两条语句
    }
}
```

对这个例子的修正很简单，只要交换子类 `Bar` 构造器中两个语句的顺序即可。但是，如果你依赖于前 ES6 支持跳过调用“父构造器”这一特性，那么要小心，因为在 ES6 中已经不允许这么做了。

3. 扩展原生类

新的 `class` 和 `extend` 设计带来的最大好处之一是（终于！）可以构建内置类的子类了。比如 `Array`。考虑：

```
class MyCoolArray extends Array {
    first() { return this[0]; }
    last() { return this[this.length - 1]; }
}

var a = new MyCoolArray( 1, 2, 3 );

a.length;                   // 3
a;                          // [1,2,3]

a.first();                  // 1
a.last();                   // 3
```

在 ES6 之前，有一个 `Array` 的伪“子类”通过手动创建对象并链接到 `Array.prototype`，只能部分工作。它不支持真正 array 的特有性质，比如自动更新 `length` 属性。ES6 子类则可以完全按照期望“继承”并新增特性！

另外一个常见的前 ES6“子类”局限是在创建自定义 error“子类”时与 `Error` 对象所相关的。真正的 `Error` 对象创建时，会自动捕获特殊的 `stack` 信息，包括生成错误时的行号和文件名。前 ES6 自定义 error“子类”没有这样的特性，这严重限制了它们的应用。

ES6 前来解救：

```
class Oops extends Error {
    constructor(reason) {
        this.oops = reason;
    }
}

// 之后：
var ouch = new Oops( "I messed up!" );
throw ouch;
```

前面代码中的自定义 error 对象 `ouch` 的行为就像其他任何真正 error 对象一样，包括捕获 `stack`。这是一个巨大的改进！

3.4.3 new.target

ES6 以 `new.target` 的形式引入了一个新概念，称为**元属性**（meta property，参见第 7 章）。

如果说看起来有点奇怪的话，那么实际上也是这样；把关键字通过 . 与属性名关联起来可绝对不是常见的 JavaScript 模式。

`new.target` 是一个新的在所有函数中都可用的“魔法”值，尽管在一般函数中它通常是 `undefined`。在任何构造器中，`new.target` 总是指向 `new` 实际上直接调用的构造器，即使构造器是在父类中且通过子类构造器用 `super(..)` 委托调用。

考虑：

```
class Foo {
    constructor() {
        console.log( "Foo: ", new.target.name );
    }
}

class Bar extends Foo {
    constructor() {
        super();
        console.log( "Bar: ", new.target.name );
    }
    baz() {
        console.log( "baz: ", new.target );
    }
}

var a = new Foo();
// Foo: Foo

var b = new Bar();
// Foo: Bar   <-- 遵循new调用点
// Bar: Bar

b.baz();
// baz: undefined
```

除了访问静态属性 / 方法（参见下一小节）之外，类构造器中的 `new.target` 元属性没有什么其他用处。

如果 `new.target` 是 `undefined`，那么你就可以知道这个函数不是通过 `new` 调用的。因此如果需要的话可以强制一个 `new` 调用。

3.4.4 static

当子类 Bar 从父类 Foo 扩展的时候，我们已经看到 Bar.prototype 是 [[Prototype]] 链接到 Foo.prototype 的。然而，Bar() 也 [[Prototype]] 链接到 Foo()。这一点可能并不显而易见。

但是，在为一个类声明了 static 方法（不只是属性）的情况下这就很有用了，因为这些是直接添加到这个类的函数对象上的，而不是在这个函数对象的 prototype 对象上。考虑：

```
class Foo {
    static cool() { console.log( "cool" ); }
    wow() { console.log( "wow" ); }
}

class Bar extends Foo {
    static awesome() {
        super.cool();
        console.log( "awesome" );
    }
    neat() {
        super.wow();
        console.log( "neat" );
    }
}

Foo.cool();                 // "cool"

Bar.cool();                 // "cool"
Bar.awesome();              // "cool"
                            // "awesome"
var b = new Bar();
b.neat();                   // "wow"
                            // "neat"

b.awesome;                  // undefined
b.cool;                     // undefined
```

小心不要误以为 static 成员在类的原型链上。实际上它们在函数构造器之间的双向 / 并行链上。

Symbol.species Getter 构造器

static 适用的一个地方就是为派生（子）类设定 Symbol.species getter（规范内称为 @@species）。如果当任何父类方法需要构造一个新实例，但不想使用子类的构造器本身时，这个功能使得子类可以通知父类应该使用哪个构造器。

举例来说，Array 有很多方法会创造并返回一个新的 Array 实例。如果定义一个 Array 的子类，但是想要这些方法仍然构造真正的 Array 实例而不是你的子类实例，就可以这样使用：

```
class MyCoolArray extends Array {
    // 强制species为父构造器
    static get [Symbol.species]() { return Array; }
```

```
}

var a = new MyCoolArray( 1, 2, 3 ),
    b = a.map( function(v){ return v * 2; } );

b instanceof MyCoolArray;   // false
b instanceof Array;         // true
```

下面说明父类方法如何使用子类 species 声明，这有点类似于 `Array#map(..)` 所做的，考虑：

```
class Foo {
    // 推迟species为子构造器
    static get [Symbol.species]() { return this; }
    spawn() {
        return new this.constructor[Symbol.species]();
    }
}

class Bar extends Foo {
    // 强制species为父构造器
    static get [Symbol.species]() { return Foo; }

}
var a = new Foo();
var b = a.spawn();
b instanceof Foo;                  // true

var x = new Bar();
var y = x.spawn();
y instanceof Bar;                  // false
y instanceof Foo;                  // true
```

父类 `Symbol.species` 通过 `return this` 来延迟到子类，就像通常期望的那样。然后 `Bar` 覆盖手动声明使用 `Foo` 来进行实例创建。当然，子类仍然可以使用 `new this.constructor(..)` 来创建自身的实例。

3.5 小结

在代码组织方面，ES6 引入了几个新特性。

- 迭代器提供了对数组或运算的顺序访问。可以通过像 `for..of` 和 `...` 这些新语言特性来消耗迭代器。
- 生成器是支持本地暂停 / 恢复的函数，通过迭代器来控制。它们可以用于编程（也是交互地，通过 `yield/next(..)` 消息传递）**生成**供迭代消耗的值。
- 模块支持对实现细节的私有封装，并提供公开导出 API。模块定义是基于文件的单例实例，在编译时静态决议。
- 类对基于原型的编码提供了更清晰的语法。新增的 `super` 也解决了 `[[Prototype]]` 链中相对引用的棘手问题。

如果想要通过拥抱 ES6 来改进你的 JavaScript 项目架构，那么应该首先考虑这些新工具。

第 4 章

异步流控制

如果你已经编写过大量 JavaScript 代码，那么一定了解异步编程是必需的技能。管理异步的主要机制一直以来都是函数回调。

然而，ES6 增加了一个新的特性来帮助解决只用回调实现异步的严重缺陷：Promise。另外，可以回顾一下（前面章节中的）生成器，研究一种组合使用这两者的模式，这是 JavaScript 异步流控制技术的一大进步。

4.1 Promise

让我们先来理清一些错误观念：Promise 不是对回调的替代。Promise 在回调代码和将要执行这个任务的异步代码之间提供了一种可靠的中间机制来管理回调。

另外一种看待 Promise 的角度是把它看作事件监听者。可以在其上注册以监听某个事件，在任务完成之后得到通知。这是一个只触发一次的事件，但仍然可以被看作事件。

可以把 Promise 链接到一起，这就把一系列异步完成的步骤串联了起来。通过与像 `all(..)` 方法（经典术语中称为“门”）和 `race(..)` 方法（经典术语中称为“latch”）这样更高级的抽象概念结合起来，Promise 链提供了一个近似的异步控制流。

还有一种定义 Promise 的方式，就是把它看作一个**未来值**（future value），对一个值的独立于时间的封装容器。不管这个容器底层的值是否已经最终确定，都可以用同样的方法应用其值。一旦观察到 Promise 的决议就立刻提取出这个值。换句话说，Promise 可以被看作是同步函数返回值的异步版本。

Promise 的决议结果只有两种可能：完成或拒绝，附带一个可选的单个值。如果 Promise 完成，那么最终的值称为完成值；如果拒绝，那么最终的值称为原因（也就是"拒绝的原因"）。Promise 只能被决议（完成或者拒绝）**一次**。之后再次试图完成或拒绝的动作都会被忽略。因此，一旦 Promise 被决议，它就是不变量，不会发生改变。

显然，看待 Promise 有各种不同的角度。没有哪一个角度是完整的，但每一种看法都提供了整体的一面。最重要的一点是，它对只用回调的异步方法给予了重大改进，即提供了有序性、可预测性和可靠性。

4.1.1 构造和使用 Promise

可以通过构造器 `Promise(..)` 构造 promise 实例：

```
var p = new Promise( function(resolve,reject){
    // ..
} );
```

提供给构造器 `Promise(..)` 的两个参数都是函数，一般称为 `resolve(..)` 和 `reject(..)`。它们是这样使用的。

- 如果调用 `reject(..)`，这个 promise 被拒绝，如果有任何值传给 `reject(..)`，这个值就被设置为拒绝的原因值。
- 如果调用 `resolve(..)` 且没有值传入，或者传入任何非 promise 值，这个 promise 就完成。
- 如果调用 `resove(..)` 并传入另外一个 promise，这个 promise 就会采用传入的 promise 的状态（要么实现要么拒绝）——不管是立即还是最终。

下面是通过 promise 重构回调函数调用的常用方法。假定你最初是使用需要能够调用 error-first 风格回调的 `ajax(..)` 工具：

```
function ajax(url,cb) {
    // 建立请求，最终会调用cb(..)
}

// ..

ajax( "http://some.url.1", function handler(err,contents){
    if (err) {
        // 处理ajax错误
    }
    else {
        // 处理contents成功情况
    }
} );
```

可以将其转化为：

```
function ajax(url) {
    return new Promise( function pr(resolve,reject){
        // 建立请求，最终会调用resolve(..)或者reject(..)
    } );
}

// ..

ajax( "http://some.url.1" )
.then(
    function fulfilled(contents){
        // 处理contents成功情况
    },
    function rejected(reason){
        // 处理ajax出错原因
    }
);
```

Promise 有一个 `then(..)` 方法，接受一个或两个回调函数作为参数。前面的函数（如果存在的话）会作为 promise 成功完成后的处理函数。第二个函数（如果存在的话）会作为 promise 被显式拒绝后的处理函数，或者在决议过程中出现错误 / 异常的情况下的处理函数。

如果某个参数被省略，或者不是一个有效的函数——通常是 `null`，那么一个默认替代函数就会被采用。默认的成功回调把完成值传出，默认的出错回调会传递拒绝原因值。

`then(null, handleRejection)` 调用的简写形式是 `catch(handleRejection)`。

`then(..)` 和 `catch(..)` 都会自动构造并返回另外一个 promise 实例，这个实例连接到接受原来的 promise 的不管是完成或拒绝处理函数（实际调用的那个）的返回值。考虑：

```
ajax( "http://some.url.1" )
.then(
    function fulfilled(contents){
        return contents.toUpperCase();
    },
    function rejected(reason){
        return "DEFAULT VALUE";
    }
)
.then( function fulfilled(data){
    // 处理来自于原来promise的处理函数的数据
} );
```

这段代码中，从 `fulfilled(..)` 或者 `rejected(..)` 返回一个立即值，然后这个值在下次事件中被第二个 `then(..)` 的 `fulfilled(..)` 接受。如果传入的是新的 promise，那么这个新的 promise 就会作为决议结果被导入和采用：

```
ajax( "http://some.url.1" )
.then(
```

```
        function fulfilled(contents){
            return ajax(
                "http://some.url.2?v=" + contents
            );
        },
        function rejected(reason){
            return ajax(
                "http://backup.url.3?err=" + reason
            );
        }
    )
    .then( function fulfilled(contents){
        // contents来自于后续的ajax(..)调用，不管是哪个调用
    } );
```

需要注意的是：第一个 `fulfilled(..)` 内部的异常（即被拒绝的 promise）**不会**导致第一个 `rejected(..)` 被调用，因为这个处理函数只响应第一个原始 promise 的决议。而第二个 promise 会接受这个拒绝，这个 promise 是在第二个 `then(..)` 上调用的。

前面的代码片段中，我们没有监听拒绝，这意味着它会默默保持这个状态等待未来的观测。如果永远不通过 `then(..)` 或 `catch(..)` 调用来观察的话，它就会一直保持未处理状态。有些浏览器开发者终端可能会监测到这些未处理拒绝并报告出来，但是这并不是可靠的保证；我们应该一直观测 promise 拒绝。

> 这里只对 Promise 的理论和行为给出了一个简要概述。关于更深入的探讨，参见本系列《你不知道的 JavaScript（中卷）》第二部分的第 3 章。

4.1.2 Thenable

`Promise(..)` 构造器的真正实例是 Promise。但还有一些类 promise 对象，称为 **thenable**，一般来说，它们也可以用 Promise 机制解释。

任何提供了 `then(..)` 函数的对象（或函数）都被认为是 thenable。Promise 机制中所有可以接受真正 promise 状态的地方，也都可以处理 thenable。

从根本上说，thenable 就是所有类 promise 值的一个通用标签，这些类 promise 不是被真正的 `Promise(..)` 构造器而是被其他系统创造出来。从这个角度来说，通常 thenable 的可靠性要低于真正的 Promise。举个例子，考虑下面这个胡作非为的 thenable：

```
var th = {
    then: function thener( fulfilled ) {
        // 每100ms调用一次fulfilled(..)，直到永远
        setInterval( fulfilled, 100 );
    }
};
```

如果接收到了这个 thenable，并通过 `th.then(..)` 把它链接起来，很可能你会吃惊地发现自己的完成处理函数会被重复调用，而正常的 Promise 应该只会决议一次。

一般来说，如果从某个其他系统接收到一个自称 promise 或者 thenable 的东西，不应该盲目信任它。在下一小节中，我们会介绍一个 ES6 Promise 包含的工具，用来帮助解决这个信任问题。

但为了更进一步理解这个问题的危害性，考虑一下：**任何**代码中的**任何**对象，当然如果是与 Promise 一起使用的话，只要定义了名为 `then()` 的方法，就可能被当作是一个 thenable，不管这个东西的目的是否与 Promise 风格的异步编码相关。

在 ES6 之前，并没有对 `then(..)` 方法名称有任何特殊保留，可以想象应该有一些实现选用了这个方法名称，而 Promise 那时候还没有出现呢。最可能出现的误用 thenable 的情况是那些使用了 `then(..)` 方法，但是并没有严格遵循 Promise 风格的异步库——确实有几个这样的“野生”库。

你的责任是，避免把可能被误认为 thenable 的值直接用于 Promise 机制。

4.1.3 Promise API

`Promise` API 还提供了一些静态方法与 Promise 一起工作。

`Promise.resolve(..)` 创建了一个决议到传入值的 promise。我们把它的工作机制与手动方法对比一下：

```
var p1 = Promise.resolve( 42 );

var p2 = new Promise( function pr(resolve){
    resolve( 42 );
} );
```

`p1` 和 `p2` 的最终行为方式是完全相同的。通过 promise 来决议也是一样：

```
var theP = ajax( .. );

var p1 = Promise.resolve( theP );

var p2 = new Promise( function pr(resolve){
    resolve( theP );
} );
```

`Promise.resolve(..)` 是一个针对上一小节中介绍的 thenable 信任问题的解决方案。对于任何还没有完全确定是可信 promise 的值，甚至它可能是立即值，都可以通过把它传给 `Promise.resolve(..)` 来规范化。如果这个值已经是可以确定的 promise 或者 thenable，它的状态 / 决议就会被直接采用，这样会避免出错。而如果它是一个立即值，那么它会被“封装”为一个真正的 promise，这样就把它的行为方式规范为异步的。

`Promise.reject(..)` 创建一个立即被拒绝的 promise，和与它对应的 `Promise(..)` 构造器一样：

```
var p1 = Promise.reject( "Oops" );

var p2 = new Promise( function pr(resolve,reject){
    reject( "Oops" );
} );
```

`resolve(..)` 和 `Promise.resolve(..)` 可以接受 promise 并接受它的状态 / 决议，而 `reject (..)` 和 `Promise.reject(..)` 并不区分接收的值是什么。所以，如果传入 promise 或 thenable 来拒绝，这个 promise / thenable 本身会被设置为拒绝原因，而不是其底层值。

`Promise.all([ .. ])` 接受一个或多个值的数组（比如，立即值、promise、thenable）。它返回一个 promise，如果所有的值都完成，这个 promise 的结果是完成；一旦它们中的某一个被拒绝，那么这个 promise 就立即被拒绝。

从这些值 /promise 开始：

```
var p1 = Promise.resolve( 42 );
var p2 = new Promise( function pr(resolve){
    setTimeout( function(){
        resolve( 43 );
    }, 100 );
} );
var v3 = 44;
var p4 = new Promise( function pr(resolve,reject){
    setTimeout( function(){
        reject( "Oops" );
    }, 10);
} );
```

让我们考虑一下 `Promise.all([ .. ])` 如何与这些值的组合合作：

```
Promise.all( [p1,p2,v3] )
.then( function fulfilled(vals){
    console.log( vals );            // [42,43,44]
} );

Promise.all( [p1,p2,v3,p4] )
.then(
    function fulfilled(vals){
        // 不会到达这里
    },
    function rejected(reason){
        console.log( reason );        // Oops
    }
);
```

`Promise.all([ .. ])` 等待所有都完成（或者第一个拒绝），而 `Promise.race([ .. ])` 等待第一个完成或者拒绝。考虑：

```
// 注意：重新设置检测值，以避免被时序问题所误导！

Promise.race( [p2,p1,v3] )
.then( function fulfilled(val){
    console.log( val );                 // 42
} );

Promise.race( [p2,p4] )
.then(
    function fulfilled(val){
        // 不会到达这里
    },
    function rejected(reason){
        console.log( reason );          // Oops
    }
);
```

`Promise.all([])` 将会立即完成（没有完成值），`Promise.race([])` 将会永远挂起。这是一个很奇怪的不一致，因此我建议，永远不要用空数组使用这些方法。

4.2 生成器 + Promise

可以把一系列 promise 以链式表达，用以代表程序的异步流控制。考虑：

```
step1()
.then(
    step2,
    step2Failed
)
.then(
    function(msg) {
        return Promise.all( [
            step3a( msg ),
            step3b( msg ),
            step3c( msg )
        ] )
    }
)
.then(step4);
```

但是，还有一种更好的方案可以用来表达异步流控制，而且从编码规范的角度来说也要比很长的 promise 链可取得多。我们可以使用在第 3 章学到的生成器来表达我们的异步流控制。

这个重要的模式需要理解一下：生成器可以 `yield` 一个 promise，然后这个 promise 可以被绑定，用其完成值来恢复这个生成器的运行。

考虑一下用生成器来表达前面代码片段的异步流控制：

```
function *main() {
    var ret = yield step1();

    try {
        ret = yield step2( ret );
    }
    catch (err) {
        ret = yield step2Failed( err );
    }

    ret = yield Promise.all( [
        step3a( ret ),
        step3b( ret ),
        step3c( ret )
    ] );

    yield step4( ret );
}
```

表面看来，这段代码似乎比前面代码中的等价 promise 链实现更冗长。但是，它提供了一种更有吸引力，同时也是更重要、更易懂、更合理且看似同步的编码风格（通过给“返回”值的 = 赋值等）。特别是由于 try..catch 出错处理可以跨过这些隐藏的异步边界。

为什么与生成器一起使用 Promise ？不用 Promise 肯定也可以实现异步生成器编码。

Promise 是一种把普通回调或者 thunk 控制反转（参见本系列《你不知道的 JavaScript（中卷)》第二部分）反转回来的可靠系统。因此，把 Promise 的可信任性与生成器的同步代码组合在一起有效解决了回调所有的重要缺陷。另外，像 Promise.all([..]) 这样的工具也是在生成器的单个 yield 步骤表达并发性的一个优秀又简洁的方法。

这个魔法是如何实现的呢？我们需要一个可以运行生成器的**运行器**（runner），接受一个 yield 出来的 promise，然后将其连接起来用以恢复生成器，方法是或者用完成成功值，或者用拒绝原因值抛出一个错误到生成器。

很多支持异步的工具 / 库都有这样的“运行器”，比如 Q.spawn(..)，以及我的 asyncquence 库的 runner(..) 插件。而这里是用一个单独的运行器来说明这个过程的工作原理：

```
function run(gen) {
    var args = [].slice.call( arguments, 1), it;

    it = gen.apply( this, args );

    return Promise.resolve()
        .then( function handleNext(value){
            var next = it.next( value );

            return (function handleResult(next){
                if (next.done) {
                    return next.value;
```

```
                }
                else {
                    return Promise.resolve( next.value )
                        .then(
                            handleNext,
                            function handleErr(err) {
                                return Promise.resolve(
                                    it.throw( err )
                                )
                                .then( handleResult );
                            }
                        );
                }
            })( next );
        } );
}
```

> 参见本系列《你不知道的 JavaScript（中卷）》第二部分，可以获取这个工具更详尽注释的版本。另外，由各种异步库提供的这种 run 工具通常比我们这里展示的更强大，功能更完善。举例来说，asynquence 的 `runner(..)` 能够处理 `yield` 出来的 promise、序列、thunk 和立即（非 promise）值，为我们提供了无限的灵活性。

所以现在运行前面代码中的 `*main()` 就这么简单：

```
run( main )
.then(
    function fulfilled(){
        // *main()成功完成
    },
    function rejected(reason){
        // 哎呀，出错了
    }
);
```

本质上说，只要代码中出现超过两个异步步骤的流控制逻辑，都可以也应该使用由 run 工具驱动的 promise-yield 生成器以异步风格表达控制流。这样可以使代码理解和维护起来更简单。

这种“yield 一个 promise 来恢复生成器”的模式将会成为一个常用模式，这个模式非常强大，下一个版本的 JavaScript 几乎肯定会引入一个新的函数类型来自动执行这种模式而无需 run 工具。我们将在第 8 章介绍 `async function`（应该是叫这个名字）。

4.3　小结

随着 JavaScript 越来越成熟以及应用越来越广泛，异步编程越发地成为核心问题。随着需求变得越来越复杂，回调也变得越来越难以胜任，直到完全崩溃。

值得高兴的是，ES6 新增了 Promise 来弥补回调的主要缺陷之一：缺少对可预测行为方式的保证。Promise 代表了来自于可能异步的任务的未来完成值，跨越同步和异步边界对行为进行规范化。

但是，Promise 与生成器的结合完全实现了重新安排异步流控制代码来消除丑陋的回调乱炖（或称“地狱”）。

现在，我们可以在各种异步库运行器的帮助下管理这些交互，而 JavaScript 最终会提供专门的语法来支持这种交互。

第 5 章
集合

对于任何 JavaScript 程序来说，对数据的结构化组合和访问都是一个关键部分。这个语言从一开始到现在，创建数据结构的主要机制一直都是数组和对象。当然，基于它们已经建立了很多作为用户库的高级数据结构。

在 ES6 中，已经把一部分最有用的（也是性能最优的！）数据结构抽象作为原生组件新增到语言之中。

这一章里，我们会从 TypedArray 开始探讨，严格说它是几年之前 ES5 时期的技术，但那时是作为 WebGL 而不是 JavaScript 组件。在 ES6 中，它已经被语言规范直接采纳，进入一级（first class）状态。

Map 就像是一个对象（键 / 值对），但是键值并非只能为字符串，而是可以使用任何值——甚至是另一个对象或 map！ Set 与数组（值的序列）类似，但是其中的值是唯一的；如果新增的值是重复的，就会被忽略。还有相应的弱（与内存 / 垃圾回收相关）版本：WeakMap 和 WeakSet。

5.1 TypedArray

正如在本系列《你不知道的 JavaScript（中卷）》第一部分中介绍的，JavaScript 拥有一组内置类型，比如 `number` 和 `string`。很容易把称为“类型数组”这样的特性想象为一个特定类型的值构成的数组，比如一个只有字符串构成的数组。

但实际上带类型的数组更多是为了使用类数组语义（索引访问等）结构化访问二进制数

据。名称中的“type(类型)”是指看待一组位序列的“视图”，本质上就是一个映射，比如是把这些位序列映射为 8 位有符号整型数组还是 16 位有符号整型数组，等等。

如何构建这样的位集合呢？这称为一个“buffer”，最直接的方法是通过 `ArrayBuffer(..)` 构造器来构造：

```
var buf = new ArrayBuffer( 32 );
buf.byteLength;                         // 32
```

现在 `buf` 就是一个二进制 buffer，长为 32 字节（256 位），预先初始化全部为 `0`。一个 buffer 本身除了查看它的 `byteLength` 属性外，并不真正支持任何其他交互。

> 有些 Web 平台功能使用或者返回数组 buffer，比如 `FileReader#readAsArrayBuffer(..)`、`XMLHttpRequest#send(..)` 和 `ImageData`(canvas data)。

而在这个数组 buffer 之上，可以放置一个“视图”，这个视图以类型数组的形式存在。考虑：

```
var arr = new Uint16Array( buf );
arr.length;                         // 16
```

`arr` 是在这个 256 位 `buf` 上映射的一个 16 位无符号整型的类型数组，也就是说你得到了 16 个元素。

5.1.1　大小端（Endianness）

理解下面这点很重要：`arr` 的映射是按照运行 JavaScript 的平台的大小端设置（大端或小端）进行的。如果二进制数据的构造是基于某个大小端配置，而解释平台的大小端配置正相反，那么这就成了一个问题。

大小端的意思是多字节数字（比如前面代码片段中创建的 16 位无符号整型）中的低字节（8 位）位于这个数字字节表示中的右侧还是左侧。

举个例子，设想一个十进制数字 `3085`，我们需要用 16 位来表示它。如果只是用一个 16 位数字容器，不管大小端配置如何，它会被表示为二进制 `0000110000001101`（十六进制 `0c0d`）。

但是如果用两个 8 位数组表示数字 `3085`，那么大小端设置就会明显影响它在内存中的存储表示：

- `0000110000001101` / `0c0d`（大端）
- `0000110100001100` / `0d0c`（小端）

如果接收到一个来自于小端系统的表示 `3085` 的位序列 `0000110100001100`，而在大端系统中

为其建立视图，得到的值将是 3340（十进制）或者 0d0c（十六进制）。

目前 Web 上最常用的是小端表示方式，但是肯定存在不采用这种方式的浏览器。了解一块二进制数据生产方和消费方的大小端属性是很重要的。

根据 MDN，这里有一个快速检测 JavaScript 大小端的方法：

```
var littleEndian = (function() {
    var buffer = new ArrayBuffer( 2 );
    new DataView( buffer ).setInt16( 0, 256, true );
    return new Int16Array( buffer )[0] === 256;
})();
```

littleEndian 的结果可能是 true 也可能是 false，对于多数浏览器来说，它应该会返回 true。这个测试方法使用了 DataView(..)，此方法比在 buffer 上建立的视图访问（getting/setting）位提供了更底层、更小粒度的控制方法。前面代码片段中 setInt16(..) 方法的第三个参数是用来通知 DataView 要使用哪种大小端配置来操作。

> 不要把数组 buffer 的底层二进制存储的大小端和给定数字在 JavaScript 程序中如何表示搞混。比如，(3085).toString(2) 返回 "110000001101"，加上前面 4 个隐去的 "0" 看起来似乎是大端表示。实际上，这个表示法是基于 16 位视图，而不是两个 8 位字节的表示。要确定 JavaScript 环境的大小端，最好的方法就是前面的 DataView 测试。

5.1.2　多视图

单个 buffer 可以关联多个视图，比如：

```
var buf = new ArrayBuffer( 2 );

var view8 = new Uint8Array( buf );
var view16 = new Uint16Array( buf );

view16[0] = 3085;
view8[0];                       // 13
view8[1];                       // 12

view8[0].toString( 16 );        // "d"
view8[1].toString( 16 );        // "c"

// 交换（就像大小端变换一样！）
var tmp = view8[0];
view8[0] = view8[1];
view8[1] = tmp;

view16[0];                      // 3340
```

这个带类数组构造器有多个签名变体。目前我们展示的情况都是传入已经存在的 buffer。然而这种形式还接受两个额外参数：`byteOffset` 和 `length`。换句话说，可以从非 `0` 的位置开始带类型数组视图，也可以不消耗整个 buffer 长度。

如果二进制数据的 buffer 包含的数据格式大小 / 位置不均匀，这种技术是非常有用的。

举例来说，考虑一个二进制 buffer，在它的起始位置有一个 2 字节数字（也就是"字"），接着是两个 1 字节数字，然后是一个 32 位浮点数。这展示了如何通过在同一个 buffer 建立多个视图，从不同的位置，以不同长度访问这个数据：

```
var first = new Uint16Array( buf, 0, 2 )[0],
    second = new Uint8Array( buf, 2, 1 )[0],
    third = new Uint8Array( buf, 3, 1 )[0],
    fourth = new Float32Array( buf, 4, 4 )[0];
```

5.1.3　带类数组构造器

除了前面一节中展示的 `(buffer,[offset, [length]])` 形式，带类数组构造器还支持以下这些形式。

- [constructor\]`(length)`：在一个新的长度为 `length` 字节的 buffer 上创建一个新的视图。
- [constructor\]`(typedArr)`：创建一个新的视图和 buffer，把 `typeArr` 视图的内容复制进去。
- [constructor\]`(obj)`：创建一个新的视图和 buffer，并在类数组或对象 `obj` 上迭代复制其内容。

ES6 提供了下面这些带类数组构造器：

- `Int8Array`（8 位有符号整型），`Uint8Array`（8 位无符号整型）
  ——`Uint8ClampedArray`（8 位无符号整型，每个值会被强制设置为在 `0-255` 内）；
- `Int16Array`（16 位有符号整型），`Uint16Array`（16 位无符号整型）；
- `Int32Array`（32 位有符号整型），`Uint32Array`（32 位无符号整型）；
- `Float32Array`（32 位浮点数，IEEE-754）；
- `Float64Array`（64 位浮点数，IEEE-754）。

带类数组构造器的实例几乎和普通原生数组完全一样。一些区别包括具有固定的长度以及值都属于某种"类型"。

然而，它们的 `prototype` 方法几乎完全一样。因此，很可能可以把它们当作普通数组使用而无需转换。

举例来说：

```
var a = new Int32Array( 3 );
a[0] = 10;
```

```
a[1] = 20;
a[2] = 30;

a.map( function(v){
    console.log( v );
} );
// 10 20 30

a.join( "-" );
// "10-20-30"
```

> 不能对 TypedArray 使用不合理的 Array.prototype 方法，比如修改器（splice(..)、push(..) 等）和 concat(..)。

要清楚 TypedArray 中的元素是限制在声明的位数大小中的。如果视图给一个 Uint8Array 的某个元素赋值为大于 8 位的值，这个值会被折回（wrap around）来适应其位宽。

这可能会引起问题，比如，如果你要把 TypedArray 中的值平方。考虑：

```
var a = new Uint8Array( 3 );
a[0] = 10;
a[1] = 20;
a[2] = 30;

var b = a.map( function(v){
    return v * v;
} );

b;              // [100, 144, 132]
```

对于值 20 和 30，平方值就会溢出。要解决这样的局限，可以使用 TypedArray#from(..) 函数：

```
var a = new Uint8Array( 3 );
a[0] = 10;
a[1] = 20;
a[2] = 30;

var b = Uint16Array.from( a, function(v){
      return v * v;
} );

b;              // [100, 400, 900]
```

关于与 TypedArray 共享的 Array.from(..) 方法的更多信息，参见 6.1.2 节，特别是“映射”那一小节解释了作为第二个参数的映射函数。

还有很有趣的一点要考虑，与普通数组一样，TypedArray 也有一个 sort(..) 方法，但是这个方法默认使用数字排序方法，而不是强制转换为字符串之后进行字母序比较 。举例来说：

```
var a = [ 10, 1, 2, ];
a.sort();                               // [1,10,2]

var b = new Uint8Array( [ 10, 1, 2 ] );
b.sort();                               // [1,2,10]
```

就像 Array#sort(..) 一样，TypedArray#sort(..) 接受了一个可选比较函数参数，它们的工作方式也是完全一样的。

5.2 Map

如果你的 JavaScript 经验丰富的话，应该会了解对象是创建无序键 / 值对数据结构 [也称为映射（map）] 的主要机制。但是，对象作为映射的主要缺点是不能使用非字符串值作为键。

举例来说，考虑：

```
var m = {};

var x = { id: 1 },
    y = { id: 2 };

m[x] = "foo";
m[y] = "bar";

m[x];               // "bar"
m[y];               // "bar"
```

这里发生了什么？ x 和 y 两个对象字符串化都是 "[object Object]"，所以 m 中只设置了一个键。

有些人通过维护平行的非字符串键数组和值数组来实现伪 map，比如：

```
var keys = [], vals = [];

var x = { id: 1 },
    y = { id: 2 };

keys.push( x );
vals.push( "foo" );

keys.push( y );
vals.push( "bar" );

keys[0] === x;                // true
vals[0];                      // "foo"

keys[1] === y;                // true
vals[1];                      // "bar"
```

当然，你不想自己维护这两个平行数组，所以可以定义一个数据结构，其中包含自动管理

低层的方法。除了必须要自己实现这些工作，这样做的主要缺点是访问的时间复杂度不再是 O(1)，而是 O(*n*)。

但在 ES6 中就不再需要这么做了！只需要使用 `Map(..)`:

```
var m = new Map():

var x = { id: 1 },
    y = { id: 2 };

m.set( x, "foo" );
m.set( y, "bar" );

m.get( x );                // "foo"
m.get( y );                // "bar"
```

这里唯一的缺点就是不能使用方括号 `[ ]` 语法设置和获取值，但完全可以使用 `get(..)` 和 `set(..)` 方法完美代替。

要从 map 中删除一个元素，不要使用 `delete` 运算符，而是要使用 `delete()` 方法：

```
m.set( x, "foo" );
m.set( y, "bar" );

m.delete( y );
```

你可以通过 `clear()` 清除整个 map 的内容。要得到 map 的长度（也就是键的个数），可以使用 `size` 属性（而不是 `length`）：

```
m.set( x, "foo" );
m.set( y, "bar" );
m.size;                       // 2

m.clear();
m.size;                       // 0
```

`Map(..)` 构造器也可以接受一个 iterable（参见 3.1 节），这个迭代器必须产生一列数组，每个数组的第一个元素是键，第二个元素是值。这种迭代的形式和 `entries()` 方法产生的形式是完全一样的，下一小节将会介绍。这使得创建一个 map 的副本很容易：

```
var m2 = new Map( m.entries() );

// 等价于：
var m2 = new Map( m );
```

因为 map 的实例是一个 iterable，它的默认迭代器与 `entries()` 相同，所以我们更推荐使用后面这个简短的形式。

当然，也可以在 `Map(..)` 构造器中手动指定一个**项目**（entry）列表（键 / 值数组的数组）：

```
var x = { id: 1 },
    y = { id: 2 };

var m = new Map( [
    [ x, "foo" ],
    [ y, "bar" ]
] );

m.get( x );                     // "foo"
m.get( y );                     // "bar"
```

5.2.1 Map 值

要从 map 中得到一列值，可以使用 `values(..)`，它会返回一个迭代器。在第 2 章和第 3 章中，我们介绍了几种顺序处理迭代器（就像数组）的方法，比如 spread 运算符 `...` 和 `for..of` 循环。另外，在 6.1.3 节我们将会详细介绍 `Array.from(..)` 方法。考虑：

```
var m = new Map();

var x = { id: 1 },
    y = { id: 2 };

m.set( x, "foo" );
m.set( y, "bar" );

var vals = [ ...m.values() ];

vals;                           // ["foo","bar"]
Array.from( m.values() );       // ["foo","bar"]
```

前面一小节介绍过，可以在一个 map 的项目上使用 `entries()` 迭代（或者默认 map 迭代器）。考虑：

```
var m = new Map();

var x = { id: 1 },
    y = { id: 2 };

m.set( x, "foo" );
m.set( y, "bar" );

var vals = [ ...m.entries() ];

vals[0][0] === x;               // true
vals[0][1];                     // "foo"

vals[1][0] === y;               // true
vals[1][1];                     // "bar"
```

5.2.2 Map 键

要得到一列键，可以使用 `keys()`，它会返回 map 中键上的迭代器：

```
var m = new Map();

var x = { id: 1 },
    y = { id: 2 };

m.set( x, "foo" );
m.set( y, "bar" );

var keys = [ ...m.keys() ];

keys[0] === x;              // true
keys[1] === y;              // true
```

要确定一个 map 中是否有给定的键，可以使用 has(..) 方法：

```
var m = new Map();

var x = { id: 1 },
    y = { id: 2 };

m.set( x, "foo" );

m.has( x );                  // true
m.has( y );                  // false
```

map 的本质是允许你把某些额外的信息（值）关联到一个对象（键）上，而无需把这个信息放入对象本身。

对于 map 来说，尽管可以使用任意类型的值作为键，但通常我们会使用对象，因为字符串或者其他基本类型已经可以作为普通对象的键使用。换句话说，除非某些或者全部键需要是对象，否则可以继续使用普通对象作为影射，这种情况下 map 才更加合适。

> 如果使用对象作为映射的键，这个对象后来被丢弃（所有的引用解除），试图让垃圾回收（GC）回收其内存，那么 map 本身仍会保持其项目。你需要从 map 中移除这个项目来支持 GC。在下一小节中，我们将会介绍作为对象键和 GC 的更好选择——WeakMap。

5.3 WeakMap

WeakMap 是 map 的变体，二者的多数外部行为特性都是一样的，区别在于内部内存分配（特别是其 GC）的工作方式。

WeakMap（只）接受对象作为键。这些对象是被**弱**持有的，也就是说如果对象本身被垃圾回收的话，在 WeakMap 中的这个项目也会被移除。然而我们无法观测到这一点，因为对象被垃圾回收的唯一方式是没有对它的引用了。但是一旦不再有引用，你也就没有对象引用来查看它是否还存在于这个 WeakMap 中了。

除此之外，WeakMap 的 API 是类似的，尽管要更少一些：

```
var m = new WeakMap();

var x = { id: 1 },
    y = { id: 2 };

m.set( x, "foo" );

m.has( x );               // true
m.has( y );               // false
```

WeakMap 没有 `size` 属性或 `clear()` 方法，也不会暴露任何键、值或项目上的迭代器。所以即使你解除了对 `x` 的引用，它将会因 GC 时这个条目被从 `m` 中移除，也没有办法确定这一事实。所以你就相信 JavaScript 所声明的吧！

和 Map 一样，通过 WeakMap 可以把信息与一个对象软关联起来。而在对这个对象没有完全控制权的时候，这个功能特别有用，比如 DOM 元素。如果作为映射键的对象可以被删除，并支持垃圾回收，那么 WeakMap 就更是合适的选择了。

需要注意的是，WeakMap 只是弱持有它的键，而不是值。考虑：

```
var m = new WeakMap();

var x = { id: 1 },
    y = { id: 2 },
    z = { id: 3 },
    w = { id: 4 };

m.set( x, y );

x = null;                   // { id: 1 } 可GC
y = null;                   // { id: 2 } 可GC
                            // 只因 { id: 1 } 可GC

m.set( z, w );

w = null;                   // { id: 4 } 不可GC
```

因此，我认为 WeakMap 更应该叫作“Weak-KeyMap”。

5.4 Set

set 是一个值的集合，其中的值唯一（重复会被忽略）。

set 的 API 和 map 类似。只是 `add(..)` 方法代替了 `set(..)` 方法（某种程度上说有点讽刺），没有 `get(..)` 方法。

考虑：

```
var s = new Set();

var x = { id: 1 },
    y = { id: 2 };

s.add( x );
s.add( y );
s.add( x );

s.size;                         // 2

s.delete( y );
s.size;                         // 1

s.clear();
s.size;                         // 0
```

Set(..) 构造器形式和 Map(..) 类似，都可以接受一个 iterable，比如另外一个 set 或者仅仅是一个值的数组。但是，和 Map(..) 接受项目（entry）列表（键 / 值数组的数组）不同，Set(..) 接受的是值（value）列表（值的数组）：

```
var x = { id: 1 },
    y = { id: 2 };

var s = new Set( [x,y] );
```

set 不需要 get(..) 是因为不会从集合中取一个值，而是使用 has(..) 测试一个值是否存在：

```
var s = new Set();

var x = { id: 1 },
    y = { id: 2 };

s.add( x );

s.has( x );             // true
s.has( y );             // false
```

除了会把 -0 和 0 当作是一样的而不加区别之外，has(..) 中的比较算法和 Object.is(..) 几乎一样（参见第 6 章）。

Set 迭代器

set 的迭代器方法和 map 一样。对于 set 来说，二者行为特性不同，但它和 map 迭代器的行为是对称的。考虑：

```
var s = new Set();

var x = { id: 1 },
    y = { id: 2 };

s.add( x ).add( y );

var keys = [ ...s.keys() ],
    vals = [ ...s.values() ],
    entries = [ ...s.entries() ];

keys[0] === x;
keys[1] === y;

vals[0] === x;
vals[1] === y;

entries[0][0] === x;
entries[0][1] === x;
entries[1][0] === y;
entries[1][1] === y;
```

keys() 和 values() 迭代器都从 set 中 yield 出一列不重复的值。entries() 迭代器 yield 出一列项目数组，其中的数组的两个项目都是唯一 set 值。set 默认的迭代器是它的 values() 迭代器。

set 固有的唯一性是它最有用的特性。举例来说：

```
var s = new Set( [1,2,3,4,"1",2,4,"5"] ),
    uniques = [ ...s ];

uniques;                        // [1,2,3,4,"1","5"]
```

set 的唯一性不允许强制转换，所以 1 和 "1" 被认为是不同的值。

5.5 WeakSet

就像 WeakMap 弱持有它的键（对其值是强持有的）一样，WeakSet 对其值也是弱持有的（这里并没有键）：

```
var s = new WeakSet();

var x = { id: 1 },
    y = { id: 2 };

s.add( x );
s.add( y );

x = null;                      // x可GC
y = null;                      // y可GC
```

WeakSet 的值必须是对象，而并不像 set 一样可以是原生类型值。

5.6 小结

ES6 定义了几个有用的集合，这使得对数据的访问更结构化且更高效。

TypedArray 提供了对二进制数据 buffer 的各种整型类型“视图”，比如 8 位无符号整型和 32 位浮点型。对二进制数据的数组访问使得运算更容易表达和维护，从而可以更容易操纵视频、音频、canvas 数据等这样的复杂数据。

Map 是键 – 值对，其中的键不只是字符串 / 原生类型，也可以是对象。Set 是成员值（任意类型）唯一的列表。

WeakMap 也是 map，其中的键（对象）是弱持有的，因此当它是对这个对象的最后一个引用的时候，GC（垃圾回收）可以回收这个项目。WeakSet 也是 set，其中的值是弱持有的，也就是说如果其中的项目是对这个对象最后一个引用的时候，GC 可以移除它。

第 6 章

新增 API

从值的转换到数学计算，ES6 为各种内置原生类型和对象新增了很多静态属性和方法，用来辅助完成一些常见的任务。另外，某些原生类型的实例通过新的原型方法有了新的功能。

> 这些特性中大多数都可以忠实 polyfill。这里我们不讨论这样的细节，你可以在“ES6 Shim”项目（https://github.com/paul millr/es6-shim/）中找到符合标准的 shim / polyfill。

6.1　Array

各种 JavaScript 用户库扩展最多的特性之一就是数组（Array）类型。所以 ES6 为 Array 增加了一些静态函数和原型（实例）方法辅助函数也在意料之中。

6.1.1　静态函数 `Array.of(..)`

`Array(..)` 构造器有一个众所周知的陷阱，就是如果只传入一个参数，并且这个参数是数字的话，那么不会构造一个值为这个数字的单个元素的数组，而是构造一个空数组，其 `length` 属性为这个数字。这个动作会产生不幸又诡异的“空槽”行为，这是 JavaScript 数组广为人所诟病的一点。

`Array.of(..)` 取代了 `Array(..)` 成为数组的推荐函数形式构造器，因为 `Array.of(..)` 并没有这个特殊的单个数字参数的问题。考虑：

```
var a = Array( 3 );
a.length;                      // 3
a[0];                          // undefined

var b = Array.of( 3 );
b.length;                      // 1
b[0];                          // 3

var c = Array.of( 1, 2, 3 );
c.length;                      // 3
c;                             // [1,2,3]
```

什么情况下你会需要使用 `Array.of(..)` 而不是只用 `c = [1,2,3]` 这样的字面值语法创建一个数组呢？有两种可能的情况。

如果你有一个回调函数需要传入的参数封装为数组，`Array.of(..)` 可以完美解决这个需求。这样的用法不是很常见，但是可能恰好"解了你的痒"。

另外一种情况是，如果你构建 `Array` 的子类（参见 3.4 节），并且想要在你的子类实例中创建和初始化元素，比如：

```
class MyCoolArray extends Array {
    sum() {
        return this.reduce( function reducer(acc,curr){
            return acc + curr;
        }, 0 );
    }
}

var x = new MyCoolArray( 3 );
x.length;                     // 3--oops!
x.sum();                      // 0--oops!

var y = [3];                  // Array，而不是MyCoolArray
y.length;                     // 1
y.sum();                      // sum不是一个函数

var z = MyCoolArray.of( 3 );
z.length;                     // 1
z.sum();                      // 3
```

你不能（简单地）只是为 `MyCoolArray` 创建一个构造器来覆盖 `Array` 父构造器的行为，因为那个构造器对于实际构造一个行为符合规范的数组值（初始化 `this`）是必要的。`MyCoolArray` 子类"继承来的"静态 `of(..)` 方法提供了很好的解决方案。

6.1.2 静态函数 Array.from(..)

JavaScript 中的"类（似）数组对象"是指一个有 `length` 属性，具体说是大于等于 0 的整数值的对象。

这样的值在使用 JavaScript 工作的过程中是非常令人沮丧的；普遍的需求就是把它们转换为真正的数组，这样就可以应用各种 `Array.prototype` 方法（`map(..)`、`indexOf(..)` 等）了。这个过程通常类似于：

```
// 类数组对象
var arrLike = {
    length: 3,
    0: "foo",
    1: "bar"
};

var arr = Array.prototype.slice.call( arrLike );
```

另外一个常见的任务是使用 `slice(..)` 来复制产生一个真正的数组：

```
var arr2 = arr.slice();
```

两种情况下，新的 ES6 `Array.from(..)` 方法都是更好理解、更优雅、更简洁的替代方法：

```
var arr = Array.from( arrLike );

var arrCopy = Array.from( arr );
```

`Array.from(..)` 检查第一个参数是否为 iterable（参见 3.1 节），如果是的话，就使用迭代器来产生值并“复制”进入返回的数组。因为真正的数组有一个这些值之上的迭代器，所以会自动使用这个迭代器。

而如果你把类数组对象作为第一个参数传给 `Array.from(..)`，它的行为方式和 `slice()`（没有参数）或者 `apply(..)` 是一样的，就是简单地按照数字命名的属性从 `0` 开始直到 `length` 值在这些值上循环。

考虑：

```
var arrLike = {
    length: 4,
    2: "foo"
};

Array.from( arrLike );
// [ undefined, undefined, "foo", undefined ]
```

因为位置 `0`、`1` 和 `3` 在 `arrLike` 上并不存在，所以在这些位置上是 `undefined` 值。

你也可以这样产生类似的结果：

```
var emptySlotsArr = [];
emptySlotsArr.length = 4;
emptySlotsArr[2] = "foo";
```

```
Array.from( emptySlotsArr );
// [ undefined, undefined, "foo", undefined ]
```

1. 避免空槽位

前面代码中的 emptySlotArr 和 Array.from(..) 调用的结果有一个微妙但重要的区别。也就是 Array.from(..) 永远不会产生空槽位。

在 ES6 之前，如果你想要产生一个初始化为某个长度，在每个槽位上都是真正的 undefined 值（不是空槽位！）的数组，不得不做额外的工作：

```
var a = Array( 4 );
// 4个空槽位!

var b = Array.apply( null, { length: 4 } );
// 4个undefined值
```

而现在 Array.from(..) 使其简单了很多：

```
var c = Array.from( { length: 4 } );
// 4个undefined值
```

像前面代码中的 a 那样使用空槽位数组能在某些数组函数上工作，但是另外一些会忽略空槽位（比如 map(..) 等）。永远不要故意利用空槽位工作，因为它几乎肯定会导致程序出现诡异 / 意料之外的行为。

2. 映射

Array.from(..) 工具还有另外一个有用的技巧。如果提供了的话，第二个参数是一个映射回调（和一般的 Array#map(..) 所期望的几乎一样），这个函数会被调用，来把来自于源的每个值映射 / 转换到返回值。考虑：

```
var arrLike = {
    length: 4,
    2: "foo"
};

Array.from( arrLike, function mapper(val,idx){
    if (typeof val == "string") {
        return val.toUpperCase();
    }
    else {
        return idx;
    }
} );
// [ 0, 1, "FOO", 3 ]
```

和其他接收回调的数组方法一样，Array.from(..) 接收一个可选的第三个参数，如果设置了的话，这个参数为作为第二个参数传入的回调指定 this 绑定。否则，this 将会是 undefined。

参见 5.1 节，其中给出了使用 Array.from(..) 把 8 位值数组转换为 16 位值数组的例子。

6.1.3　创建数组和子类型

前面几小节中，我们已经讨论了 Array.of(..) 和 Array.from(..)，二者都以与构造器类似的方式创建一个新数组，而在子类型方面它们又是怎样的呢？它们会创建基类 Array 的实例还是继承子类型的实例呢？

```
class MyCoolArray extends Array {
    ..
}

MyCoolArray.from( [1, 2] ) instanceof MyCoolArray;  // true

Array.from(
    MyCoolArray.from( [1, 2] )
) instanceof MyCoolArray;                           // false
```

of(..) 和 from(..) 都使用访问它们的构造器来构造数组。所以如果使用基类 Array.of(..)，那么得到的就是 Array 实例；如果使用 MyCoolArray.of(..)，那么得到的就是 MyCoolArray 实例。

在 3.4 节中，我们介绍了 @@species 设置，所有的内置类（比如 Array）都有定义，任何创建新实例的原型方法都会使用它。slice(..) 是一个很好的例子：

```
var x = new MyCoolArray( 1, 2, 3 );

x.slice( 1 ) instanceof MyCoolArray;                // true
```

一般来说，默认的行为方式很可能就是需要的，但就像我们在第 3 章中介绍的，必要的话也可以覆盖它：

```
class MyCoolArray extends Array {
    // 强制species为父构造器
    static get [Symbol.species]() { return Array; }
}

var x = new MyCoolArray( 1, 2, 3 );

x.slice( 1 ) instanceof MyCoolArray;          // false
x.slice( 1 ) instanceof Array;                // true
```

需要注意的是，@@species 设置只用于像 slice(..) 这样的原型方法。of(..) 和 from(..)

不会使用它；它们都只使用 this 绑定（由使用的构造器来构造其引用）。考虑：

```
class MyCoolArray extends Array {
    // 强制species为父构造器
    static get [Symbol.species]() { return Array; }
}

var x = new MyCoolArray( 1, 2, 3 );

MyCoolArray.from( x ) instanceof MyCoolArray;        // true
MyCoolArray.of( [2, 3] ) instanceof MyCoolArray;     // true
```

6.1.4　原型方法 copyWithin(..)

Array#copyWithin(..) 是一个新的修改器方法，（包括带类型的数组在内的，参见第 5 章）所有数组都支持。copyWithin(..) 从一个数组中复制一部分到同一个数组的另一个位置，覆盖这个位置所有原来的值。

参数是 target（要复制到的索引）、start（开始复制的源索引，包括在内）以及可选的 end（复制结束的不包含索引）。如果任何一个参数是负数，就被当作是相对于数组结束的相对值。

考虑：

```
[1,2,3,4,5].copyWithin( 3, 0 );          // [1,2,3,1,2]

[1,2,3,4,5].copyWithin( 3, 0, 1 );       // [1,2,3,1,5]

[1,2,3,4,5].copyWithin( 0, -2 );         // [4,5,3,4,5]

[1,2,3,4,5].copyWithin( 0, -2, -1 );     // [4,2,3,4,5]
```

就像前面代码片段展示的，copyWithin(..) 方法不会增加数组的长度。到达数组结尾复制就会停止。

与你想象的正相反，复制并非总是从左到右（索引递增）进行的。如果源范围和目标范围重叠的话，可能会出现重复复制已经复制的值，而这可能并非你想要的结果。

所以，内部算法通过反向复制避免了这种情况。考虑：

```
[1,2,3,4,5].copyWithin( 2, 1 );     // ???
```

如果算法严格按照从左到右来移动，那么 2 应该被复制来覆盖 3，然后这个被复制的 2 应该被复制来覆盖 4，然后这个被复制的 2 应该被复制来覆盖 5，而你最终会得到 [1,2,2,2,2]。

而实际上，复制算法会反向进行，复制 4 来覆盖 5，然后复制 3 来覆盖 4，然后复制 2 来覆盖 3，最后的结果是 [1,2,2,3,4]。根据期望来说，这可能是更“正确”的结果，但如果只

考虑简单的从左到右方式的复制算法，你可能会觉得很迷惑。

6.1.5 原型方法 fill(..)

可以通过 ES6 原生支持的方法 Array#fill(..) 用指定值完全（或部分）填充已存在的数组：

```
var a = Array( 4 ).fill( undefined );
a;
// [undefined,undefined,undefined,undefined]
```

fill(..) 可选地接收参数 start 和 end，它们指定了数组要填充的子集位置，比如：

```
var a = [ null, null, null, null ].fill( 42, 1, 3 );

a;                                  // [null,42,42,null]
```

6.1.6 原型方法 find(..)

一般来说，在数组中搜索一个值的最常用方法一直是 indexOf(..) 方法，这个方法返回找到值的索引，如果没有找到就返回 -1：

```
var a = [1,2,3,4,5];

(a.indexOf( 3 ) != -1);             // true
(a.indexOf( 7 ) != -1);             // false

(a.indexOf( "2" ) != -1);           // false
```

相比之下，indexOf(..) 需要严格匹配 ===，所以搜索 "2" 不会找到值 2，反之也是如此。indexOf(..) 的匹配算法无法覆盖，而且要手动与值 -1 进行比较也很麻烦 / 笨拙。

> 本系列《你不知道的 JavaScript（中卷）》第一部分中给出了一个有趣（也充满争议地难以理解）的技术，用 ~ 运算符来绕过丑陋的返回值 -1 的问题。

从 ES5 以来，控制匹配逻辑的最常用变通技术是使用 some(..) 方法。它的实现是通过为每个元素调用一个函数回调，直到某次调用返回 true / 真值时才会停止。因为你可以定义这个回调函数，也就有了对匹配方式的完全控制：

```
var a = [1,2,3,4,5];

a.some( function matcher(v){
    return v == "2";
} );                    // true
```

```
a.some( function matcher(v){
    return v == 7;
} );                    // false
```

但这种方式的缺点是如果找到匹配的值的时候，只能得到匹配的 true/false 指示，而无法得到真正的匹配值本身。

ES6 的 find(..) 解决了这个问题。基本上它和 some(..) 的工作方式一样，除了一旦回调返回 true/ 真值，会返回实际的数组值：

```
var a = [1,2,3,4,5];

a.find( function matcher(v){
    return v == "2";
} );                                // 2

a.find( function matcher(v){
    return v == 7;                  // undefined
});
```

通过自定义 matcher(..) 函数也可以支持比较像对象这样的复杂值：

```
var points = [
    { x: 10, y: 20 },
    { x: 20, y: 30 },
    { x: 30, y: 40 },
    { x: 40, y: 50 },
    { x: 50, y: 60 }
];

points.find( function matcher(point) {
    return (
        point.x % 3 == 0 &&
        point.y % 4 == 0
    );
} );                                // { x: 30, y: 40 }
```

就像其他接受回调的数组方法一样，find(..) 接受一个可选的第二个参数，如果设定这个参数就绑定到第一个参数回调的 this。否则，this 就是 undefined。

6.1.7 原型方法 findIndex(..)

前面一小节展示了 some(..) 如何 yield 出一个布尔型结果用于在数组中搜索，以及 find(..) 如何从数组搜索 yield 出匹配的值本身，另外，还需要找到匹配值的位置索引。

indexOf(..) 会提供这些，但是无法控制匹配逻辑；它总是使用 === 严格相等。所以 ES6 的 findIndex(..) 才是解决方案：

```
var points = [
    { x: 10, y: 20 },
    { x: 20, y: 30 },
    { x: 30, y: 40 },
    { x: 40, y: 50 },
    { x: 50, y: 60 }
];

points.findIndex( function matcher(point) {
    return (
        point.x % 3 == 0 &&
        point.y % 4 == 0
    );
} );                                // 2

points.findIndex( function matcher(point) {
    return (
        point.x % 6 == 0 &&
        point.y % 7 == 0
    );
} );                                // -1
```

不要使用 `findIndex(..) != -1`（这是 `indexOf(..)` 的惯用法）从搜索中得到布尔值，因为 `some(..)` 已经 yield 出你想要的 `true/false`。也不要用 `a[ a.findIndex(..) ]` 来得到匹配值，因为这是 `find(..)` 所做的事。最后，如果需要严格匹配的索引值，那么使用 `indexOf(..)`；如果需要自定义匹配的索引值，那么使用 `findIndex(..)`。

> 就像其他接收回调的数组方法一样，`findIndex(..)` 接收一个可选的第二个参数，如果设定这个参数就绑定到第一个参数回调的 `this`。否则，`this` 就是 `undefined`。

6.1.8　原型方法 entries()、values()、keys()

在第 3 章中，我们展示了各种数据结构如何通过迭代器提供一个依次枚举其值的模式。然后在第 5 章探索新的 ES6 集合（Map、Set 等）如何提供了几种方法以产生不同迭代的时候，详细展示了这种方法。

因为 `Array` 对于 ES6 来说已经不是新的了，所以从传统角度来说，它可能不会被看作是“集合”，但是它提供了同样的迭代器方法 `entries()`、`values()` 和 `keys()`，从这个意义上说，它是一个集合。考虑：

```
var a = [1,2,3];

[...a.values()];                    // [1,2,3]
[...a.keys()];                      // [0,1,2]
[...a.entries()];                   // [ [0,1], [1,2], [2,3] ]

[...a[Symbol.iterator]()];          // [1,2,3]
```

就像 Set 一样，默认的 Array 迭代器和 values() 返回的值一样。

在后面的 6.5.4 节中，我们将展示 Array.from(..) 如何把数组中的空槽位看作值为 undefined 的槽位。这实际上是因为在底层数组迭代器是这样工作的：

```
var a = [];
a.length = 3;
a[1] = 2;

[...a.values()];        // [undefined,2,undefined]
[...a.keys()];          // [0,1,2]
[...a.entries()];       // [ [0,undefined], [1,2], [2,undefined] ]
```

6.2 Object

Object 也新增了几个静态辅助函数。传统上认为，这一类函数的关注点在对象值的行为方式 / 功能上。

但是，从 ES6 开始，Object 静态方法也开始用于那些还没有更自然的有另外归属的（比如 Array.from(..)) 通用全局 API。

6.2.1 静态函数 Object.is(..)

静态函数 Object.is(..) 执行比 === 比较更严格的值比较。

Object.is(..) 调用底层 SameValue 算法（ES6 规范，7.2.9 节）。SameValue 算法基本上和 === 严格相等比较算法一样（ES6 规范，7.2.13 节），但有两个重要的区别。

考虑：

```
var x = NaN, y = 0, z = -0;

x === x;                    // false
y === z;                    // true

Object.is( x, x );          // true
Object.is( y, z );          // false
```

你应该继续使用 === 进行严格相等比较；不应该把 Object.is(..) 当作这个运算符的替代。但是，如果需要严格识别 NaN 或者 -0 值，那么应该选择 Object.is(..)。

> ES6 还新增了一个 Number.isNaN(..) 工具（本章后面会介绍），这个工具可能是更方便的检查工具；与 Object.is(x,NaN) 相比，你可能更喜欢 Number.isNaN(x)。你可以使用 x == 0 && 1 / x === -Infinity 这种笨拙的方式精确判断 -0 值，但这种情况下使用 Object.is(x,-0) 会好很多。

6.2.2　静态函数 Object.getOwnPropertySymbols(..)

我们在 2.13 节讨论过 ES6 中新增的基本值类型 Symbol。

Symbol 很可能会成为对象最常用的特殊（元）属性。所以引入了工具 Object.getOwnPropertySymbols(..)，它直接从对象上取得所有的符号属性：

```
var o = {
    foo: 42,
    [ Symbol( "bar" ) ]: "hello world",
    baz: true
};

Object.getOwnPropertySymbols( o );  // [ Symbol(bar) ]
```

6.2.3　静态函数 Object.setPrototypeOf(..)

还是在第 2 章中，我们提到工具 Object.setPrototypeOf(..)，这个工具（不出人意料地）设置对象的 [[Prototype]] 用于**行为委托**（参见本系列《你不知道的 JavaScript（上卷）》第二部分）。考虑：

```
var o1 = {
    foo() { console.log( "foo" ); }
};
var o2 = {
    // .. o2的定义 ..
};

Object.setPrototypeOf( o2, o1 );

// 委托给o1.foo()
o2.foo();                           // foo
```

也可以：

```
var o1 = {
    foo() { console.log( "foo" ); }
};

var o2 = Object.setPrototypeOf( {
    // .. o2的定义 ..
}, o1 );

// 委托给o1.foo()
o2.foo();                          // foo
```

前面两段代码中，o2 和 o1 的关系都出现在 o2 定义的结尾处。更通俗地说，o2 和 o1 的关系在 o2 的定义上指定，就像类一样，也和字面值对象中的 __proto__ 一样（参见 2.6.4 节）。

如前所示，紧接对象创建之后设定 [[Prototype]] 是合理的。但是在很久之后才修改它不是一个好主意，因为这通常会产生令人迷惑而非清晰的代码。

6.2.4　静态函数 Object.assign(..)

很多 JavaScript 库 / 框架提供了用于把一个对象的属性复制 / 混合到另一个对象中的工具（比如，JQuery 的 extent(..)）。这些不同的工具之间有各种细微的区别，比如是否忽略值为 undefined 的属性。

ES6 新增了 Object.assign(..)，这是这些算法的简化版本。第一个参数是 target，其他传入的参数都是源，它们将按照列出的顺序依次被处理。对于每个源来说，它的可枚举和自己拥有的（也就是不是“继承来的”）键值，包括符号都会通过简单 = 赋值被复制。Object.assign(..) 返回目标对象。

考虑这个对象设定：

```
var target = {},
    o1 = { a: 1 }, o2 = { b: 2 },
    o3 = { c: 3 }, o4 = { d: 4 };

// 设定只读属性
Object.defineProperty( o3, "e", {
    value: 5,
    enumerable: true,
    writable: false,
    configurable: false
} );

// 设定不可枚举属性
Object.defineProperty( o3, "f", {
    value: 6,
    enumerable: false
} );

o3[ Symbol( "g" ) ] = 7;

// 设定不可枚举符号
Object.defineProperty( o3, Symbol( "h" ), {
    value: 8,
    enumerable: false
} );

Object.setPrototypeOf( o3, o4 );
```

只有属性 a、b、c、e 以及 Symbol("g") 会被复制到 target 中：

```
Object.assign( target, o1, o2, o3 );

target.a;                          // 1
target.b;                          // 2
target.c;                          // 3

Object.getOwnPropertyDescriptor( target, "e" );
// { value: 5, writable: true, enumerable: true,
//   configurable: true }

Object.getOwnPropertySymbols( target );
// [Symbol("g")]
```

复制过程会忽略属性 `d`、`f` 和 `Symbol("h")`；不可枚举的属性和非自有的属性都被排除在赋值过程之外。另外，`e` 作为一个普通属性赋值被复制，而不是作为只读属性复制。

在前面一节中，我们展示了使用 `setPrototypeOf(..)` 设定对象 `o2` 和 `o1` 之间的 `[[Prototype]]` 关系。还有另外一种应用了 `Object.assign(..)` 的形式：

```
var o1 = {
    foo() { console.log( "foo" ); }
};

var o2 = Object.assign(
    Object.create( o1 ),
    {
        // .. o2的定义 ..
    }
);

// 委托给o1.foo()
o2.foo();                  // foo
```

> `Object.create(..)` 是 ES5 工具，创建一个 `[[Prototype]]` 链接的空对象。参见本系列《你不知道的 JavaScript（上卷）》第二部分获取更多信息。

6.3 Math

ES6 增加了几个新的数学工具，填补了常用计算方面的空白。这些工具都可以手动计算，但是其中多数现在有了原生定义，所以某些情况下 JavaScript 引擎可以更优化地执行这些计算，或者执行比手动版本精度更高的计算。

这些工具的使用者更可能是 asm.js/transpile 的 JavaScript 代码（参见本系列《你不知道的 JavaScript（中卷）》第二部分），而非直接开发者。

三角函数：

`cosh(..)`
　双曲余弦函数

`acosh(..)`
　双曲反余弦函数

`sinh(..)`
　双曲正弦函数

`asinh(..)`
　双曲反正弦函数

`tanh(..)`
　双曲正切函数

`atanh(..)`
　双曲反正切函数

`hypot(..)`
　平方和的平方根（也即：广义勾股定理）

算术：

`cbrt(..)`
　立方根

`clz32(..)`
　计算 32 位二进制表示的前导 0 个数

`expm1(..)`
　等价于 `exp(x) - 1`

`log2(..)`
　二进制对数（以 2 为底的对数）

`log10(..)`
　以 10 为底的对数

`log1p(..)`
　等价于 `log(x + 1)`

`imul(..)`
　两个数字的 32 位整数乘法

元工具：

`sign(..)`
　返回数字符号

`trunc(..)`
　返回数字的整数部分

`fround(..)`
　向最接近的 32 位（单精度）浮点值取整

6.4 Number

更重要的是，要让程序正常工作，必须精确处理数字。ES6 新增了额外的属性和函数来提供常用数字运算。

对 `Number` 的两个新增内容就是指向已有的全局函数的引用：`Number.parseInt(..)` 和 `Number.parseFloat(..)`。

6.4.1 静态属性

ES6 新增了一些作为静态属性的辅助数字常量：

`Number.EPSILON`
　任意两个值之间的最小差：`2^-52`（参见本系列《你不知道的 JavaScript（中卷）》第一部分的第 2 章，其使用了这个值作为浮点数算法的精度误差值）

`Number.MAX_SAFE_INTEGER`
　JavaScript 可以用数字值无歧义“安全”表达的最大整数：`2^53 - 1`

`Number.MIN_SAFE_INTEGER`
　JavaScript 可以用数字值无歧义“安全”表达的最小整数：`-(2^53 - 1)` 或 `(-2)^53 + 1`

> 关于“安全”整型的更多信息，参见本系列《你不知道的 JavaScript（中卷）》第一部分的第 2 章。

6.4.2 静态函数 Number.isNaN(..)

标准全局工具 `isNaN(..)` 自出现以来就是有缺陷的，它对非数字的东西都会返回 `true`，而不是只对真实的 `NaN` 值返回 `true`，因为它把参数强制转换为数字类型（可能会错误地导致

NaN)。ES6 增加了一个修正工具 Number.isNaN(..)，可以按照期望工作：

```
var a = NaN, b = "NaN", c = 42;

isNaN( a );                     // true
isNaN( b );                     // true--oops!
isNaN( c );                     // false

Number.isNaN( a );              // true
Number.isNaN( b );              // false--修正了!
Number.isNaN( c );              // false
```

6.4.3　静态函数 Number.isFinite(..)

看到像 isFinite(..) 这样的函数名，我们常常认为它的意思就是“非无限的”。但是这并不完全正确。这个 ES6 新工具有一些微妙之处。考虑：

```
var a = NaN, b = Infinity, c = 42;

Number.isFinite( a );           // false
Number.isFinite( b );           // false

Number.isFinite( c );           // true
```

标准的全局 isFinite(..) 会对参数进行强制类型转换，但是 Number.isFinite(..) 会略去这种强制行为：

```
var a = "42";

isFinite( a );                  // true
Number.isFinite( a );           // false
```

可能你更需要类型转换，这种情况下全局 isFinite(..) 是一个有效的选择。另外，也许是更合理的，可以使用 Number.isFinite(+x)，它会在传入之前显式地把 x 强制转换为数字（参见本系列《你不知道的 JavaScript（中卷）》第一部分的第 4 章）。

6.4.4　整型相关静态函数

JavaScript 的数字值永远都是浮点数（IEE-754）。所以确定数字是否为“整型”的概念并不是检查其类型，因为 JavaScript 并没有这样区分。

相反，你需要检查这个值的小数部分是否非 0。最简单的实现方法通常是：

```
x === Math.floor( x );
```

ES6 新增了一个辅助工具 Number.isInteger(..)，这个工具可能会更有效地确定这个性质：

```
Number.isInteger( 4 );            // true
Number.isInteger( 4.2 );          // false
```

在 JavaScript 中，4、4.、4.0 或者 4.0000 之间并没有区别。所有这些都会被当作“整型”并且从 Number.isInteger(..) 中返回 true。

另外，Number.isInteger(..) 会过滤掉 x === Math.floor(x) 可能会搞混的明显非整数值：

```
Number.isInteger( NaN );            // false
Number.isInteger( Infinity );       // false
```

使用“整数”工作有时候是一个重要的信息，因为这可能会简化某些类型的算法。JavaScript 代码本身不会因为只使用整数而运行得更快，但是，只有在使用整数时，引擎才可以采用优化技术（比如 asm.js）。

基于 Number.isInteger(..) 对 NaN 和 Infinity 值的处理方式，要定义一个 isFloat(..) 工具并不像 !Number.isInteger(..) 那么简单。可能需要做类似于下面这样的事情：

```
function isFloat(x) {
    return Number.isFinite( x ) && !Number.isInteger( x );
}

isFloat( 4.2 );             // true
isFloat( 4 );               // false

isFloat( NaN );             // false
isFloat( Infinity );        // false
```

看起来可能有点奇怪，但是 Infinity 既不应该被当作整型又不应该被当作浮点型。

ES6 还定义了一个工具 Number.isSafeInteger(..)，这个工具检查一个值以确保其为整数并且在 Number.MIN_SAFE_INTEGER-Number.MAX_SAFE_INTEGER 范围之内（全包含）：

```
var x = Math.pow( 2, 53 ),
    y = Math.pow( -2, 53 );

Number.isSafeInteger( x - 1 );          // true
Number.isSafeInteger( y + 1 );          // true

Number.isSafeInteger( x );              // false
Number.isSafeInteger( y );              // false
```

6.5 字符串

在 ES6 之前已经有了很多字符串辅助函数，现在又增加了一些。

6.5.1　Unicode 函数

我们在 2.12.1 节详细介绍过 `String.fromCodePoint(..)`、`String#codePointAt(..)` 和 `String#normalize(..)`。新增这些函数是为了提高 JavaScript 字符串值对 Unicode 的支持：

```
String.fromCodePoint( 0x1d49e );          // "𝒞"
"ab𝒞d.codePointAt( 2 ).toString( 16 );    // "1d49e"
```

字符串原型方法 `normalize(..)` 用于执行 Unicode 规范化，或者把字符用“合并符”连接起来或者把合并的字符解开。

一般来说，规范化不会对字符串的内容造成可见的效果，但是会改变字符串的内容，这可能会影响像 `length` 属性的结果，以及通过位置访问字符的方式：

```
var s1 = "e\u0301";
s1.length;                          // 2

var s2 = s1.normalize();
s2.length;                          // 1
s2 === "\xE9";                      // true
```

`normalize(..)` 接受一个可选的参数，来指定要使用的规范化形式。这个参数必须是这几个值之一：`"NFC"`（默认）、`"NFD"`、`"NFKC"` 或者 `"NFKD"`。

> 规范化形式及其对字符串产生的影响超出了本部分的讨论范围。参见“Unicode Normalization Forms”（http://www.unicode.org/reports/tr15/）获取更多信息。

6.5.2　静态函数 `String.raw(..)`

`String.raw(..)` 工具作为内置标签函数提供，与模板字符串字面值（参见第 2 章）一起使用，用于获得不应用任何转义序列的原始字符串。

这个函数基本上不会被手动调用，而是与标签模板字面值一起使用：

```
var str = "bc";

String.raw`\ta${str}d\xE9`;
// "\tabcd\xE9", 而不是"  abcdé"
```

在结果字符串中，`\` 和 `t` 是独立的原始字符，而不是转义字符 `\t`。对于 Unicode 转义序列也是一样。

6.5.3　原型函数 `repeat(..)`

像 Python 和 Ruby 这样的语言中，可以这样重复字符串：

```
"foo" * 3;                          // "foofoofoo"
```

JavaScript 不支持这种形式，因为乘法 * 只对数字有定义，因此 "foo" 被强制转换成了数字 NaN。

然而，ES6 定义了一个字符串原型方法 repeat(..) 来完成这个任务：

```
"foo".repeat( 3 );                  // "foofoofoo"
```

6.5.4　字符串检查函数

除了 ES6 之前的 String#indexOf(..) 和 String#lastIndexOf(..)，又新增了 3 个用于搜索 / 检查的新方法：startsWith(..)、endsWidth(..) 和 includes(..)。

```
var palindrome = "step on no pets";

palindrome.startsWith( "step on" );    // true
palindrome.startsWith( "on", 5 );      // true

palindrome.endsWith( "no pets" );      // true
palindrome.endsWith( "no", 10 );       // true

palindrome.includes( "on" );           // true
palindrome.includes( "on", 6 );        // false
```

对于所有的字符串搜索 / 检查方法，如果寻找空字符串 ""，总是会在字符串的开头或结尾找到。

> 默认情况下，这些方法不会接受正则表达式用于字符串搜索。参见 7.3.5 节中关于关闭对第一个参数执行的 isRegExp 检查的部分。

6.6　小结

ES6 为各种内置原生对象新增了许多额外的辅助 API。

- Array 新增了静态函数 of(..) 和 from(..)，以及像 copyWithin(..) 和 fill(..) 这样的原型函数。
- Object 新增了静态函数 is(..) 和 assign(..)。
- Math 新增了静态函数 acosh(..) 和 clz32(..)。
- Number 新增了静态属性 Number.EPSILON，以及静态函数 Number.isFinite(..)。
- String 新增了静态函数 String.fromCodePoint(..) 和 String.raw(..)，以及原型函数 repeat(..) 和 includes(..)。

这些新增内容多数都可以 polyfill（参见 ES6 Shim），其思路来自于常用的 JavaScript 库和框架。

第 7 章

元编程

元编程是指操作目标是程序本身的行为特性的编程。换句话说，它是对程序的编程的编程。有点拗口，是吧？

举例来说，如果想要查看对象 `a` 和另外一个对象 `b` 的关系是否是 `[[Prototype]]` 链接的，可以使用 `a.isPrototype(b)`，这是一种元编程形式，通常称为内省（introspection）。另外一个明显的元编程例子是宏（在 JavaScript 中还不支持）——代码在编译时修改自身。用 `for..in` 循环枚举对象的键，或者检查一个对象是否是某个“类构造器”的**实例**，也都是常见的元编程例子。

元编程关注以下一点或几点：代码查看自身、代码修改自身、代码修改默认语言特性，以此影响其他代码。

元编程的目标是利用语言自身的内省能力使代码的其余部分更具描述性、表达性和灵活性。因为元编程的**元**（meta）本质，我们有点难以给出比上面提到的更精确的定义。要理解元编程，最好的方法是通过实例来展示。

ES6 在 JavaScript 现有的基础上为元编程新增了一些形式 / 特性。

7.1 函数名称

你的代码在有些情况下可能想要了解自身，想要知道某个函数的名称是什么。如何得知一个函数的名称，答案出人意料地有些模棱两可。 考虑：

```
function daz() {
    // ..
}

var obj = {
    foo: function() {
         // ..
    },
    bar: function baz() {
        // ..
    },
    bam: daz,
    zim() {
        // ..
    }
};
```

在前面的代码中，“obj.foo() 的名称是什么”略显微妙。是 "foo"、""，还是 undefined？obj.bar() 呢？它的名称是 "bar" 还是 "baz"？ obj.bam() 的名称是 "bam" 还是 "daz"？ obj.zim() 呢？

此外，作为回调传递的函数又是怎样的呢？比如：

```
function foo(cb) {
    // 这里cb()的名称是什么？
}

foo( function(){
    // 我是匿名的！
} );
```

程序中有多种方式可以表达一个函数，函数的“名称”应该是什么并非总是清晰无疑的。

更重要的是，我们需要确定函数的“名称”是否就是它的 name 属性（是的，函数有一个名为 name 的属性），或者它是否指向其词法绑定名称，比如 function bar() {..} 中的 bar。

词法绑定名称是用于像递归这样的任务：

```
function foo(i) {
    if (i < 10) return foo( i * 2 );
    return i;
}
```

name 属性是用于元编程目的的，所以它是这里讨论的关注点。

这里比较混乱，因为默认情况下函数的词法名称（如果有的话）也会被设为它的 name 属性。实际上，ES5（和之前的）规范对这一行为并没有正式要求。name 属性的设定是非标准的，但还是比较可靠的。而在 ES6 中这一点已经得到了标准化。

如果函数设定了 name 值，那么这个值通常也就是开发者工具中栈踪迹使用的名称。

推导

对于没有词法名称的函数，name 属性是怎样的呢？

在 ES6 中，现在已经有了一组推导规则可以合理地为函数的 name 属性赋值，即使这个函数并没有词法名称可用。

考虑：

```
var abc = function() {
    // ..
};

abc.name;                   // "abc"
```

如果给了这个函数一个词法名称，比如 abc = function def() { .. }，那么 name 属性当然就是 "def"。而如果没有词法名称的话，直觉上看似乎名称为 "abc" 比较合理。

下面是 ES6 中名称推导（或者没有名称）的其他几种形式：

```
(function(){ .. });                     // name:
(function*(){ .. });                    // name:
window.foo = function(){ .. };          // name:

class Awesome {
    constructor() { .. }                // name: Awesome
    funny() { .. }                      // name: funny
}

var c = class Awesome { .. };           // name: Awesome

var o = {
    foo() { .. },                       // name: foo
    *bar() { .. },                      // name: bar
    baz: () => { .. },                  // name: baz
    bam: function(){ .. },              // name: bam
    get qux() { .. },                   // name: get qux
    set fuz() { .. },                   // name: set fuz
    ["b" + "iz"]:
        function(){ .. },               // name: biz
    [Symbol( "buz" )]:
        function(){ .. }                // name: [buz]
};

var x = o.foo.bind( o );                // name: bound foo
(function(){ .. }).bind( o );           // name: bound
```

```
export default function() { .. }     // name: default

var y = new Function();              // name: anonymous
var GeneratorFunction =
    function*(){}.__proto__.constructor;
var z = new GeneratorFunction();     // name: anonymous
```

默认情况下，name 属性不可写，但可配置，也就是说如果需要的话，可使用 Object.defineProperty(..) 来手动修改。

7.2 元属性

在 3.4.3 节中，我们介绍了 ES6 中的一个 JavaScript 新概念：元属性。正如其名称所暗示的，元属性以属性访问的形式提供特殊的其他方法无法获取的元信息。

以 new.target 为例，关键字 new 用作属性访问的上下文。显然，new 本身并不是一个对象，因此这个功能很特殊。而在构造器调用（通过 new 触发的函数 / 方法）内部使用 new.target 时，new 成了一个虚拟上下文，使得 new.target 能够指向调用 new 的目标构造器。

这个是元编程操作的一个明显示例，因为它的目的是从构造器调用内部确定最初 new 的目标是什么，通用地说就是用于内省（检查类型 / 结构）或者静态属性访问。

举例来说，你可能需要在构造器内部根据是直接调用还是通过子类调用采取不同的动作：

```
class Parent {
    constructor() {
        if (new.target === Parent) {
            console.log( "Parent instantiated" );
        }
        else {
            console.log( "A child instantiated" );
        }
    }
}

class Child extends Parent {}

var a = new Parent();
// Parent instantiated

var b = new Child();
// A child instantiated
```

这里有点微妙，Parent 类定义内部的 constructor() 实际上被给定了类的词法名称（Parent），即使语法暗示这个类是与构造器分立的实体。

对于所有的元编程技术都要小心，不要编写过于机灵的代码，让未来的你或者其他代码维护者难以理解。要小心使用这些技巧。

7.3 公开符号

在 2.13 节中，我们介绍了 ES6 新原生类型 `symbol`。除了在自己的程序中定义符号之外，JavaScript 预先定义了一些内置符号，称为**公开符号**（Well-Known Symbol，WKS）。

定义这些符号主要是为了提供专门的元属性，以便把这些元属性暴露给 JavaScript 程序以获取对 JavaScript 行为更多的控制。

接下来我们会简单介绍一下每个符号和它们的作用。

7.3.1 Symbol.iterator

在第 2 章和第 3 章中，我们介绍并使用了符号 `@@iterator`，它是由 `...` 展开和 `for..of` 循环自动使用的。在第 5 章中，我们还看到了 ES6 新特性集合中的 `@@iterator`。

`Symbol.iterator` 表示任意对象上的一个专门位置（属性），语言机制自动在这个位置上寻找一个方法，这个方法构造一个迭代器来消耗这个对象的值。很多对象定义有这个符号的默认值。

然而，也可以通过定义 `Symbol.iterator` 属性为任意对象值定义自己的迭代器逻辑，即使这会覆盖默认的迭代器。这里的元编程特性在于我们定义了一个行为特性，供 JavaScript 其他部分（也就是运算符和循环结构）在处理定义的对象时使用。

考虑：

```
var arr = [4,5,6,7,8,9];

for (var v of arr) {
    console.log( v );
}
// 4 5 6 7 8 9

// 定义一个只在奇数索引值产生值的迭代器
arr[Symbol.iterator] = function*() {
    var idx = 1;
    do {
        yield this[idx];
    } while ((idx += 2) < this.length);
};

for (var v of arr) {
```

```
    console.log( v );
}
// 5 7 9
```

7.3.2 Symbol.toStringTag 与 Symbol.hasInstance

最常见的一个元编程任务，就是在一个值上进行内省来找出它是什么种类，这通常是为了确定其上适合执行何种运算。对于对象来说，最常用的内省技术是 toString() 和 instanceof。

考虑：

```
function Foo() {}

var a = new Foo();

a.toString();               // [object Object]
a instanceof Foo;           // true
```

在 ES6 中，可以控制这些操作的行为特性：

```
function Foo(greeting) {
    this.greeting = greeting;
}

Foo.prototype[Symbol.toStringTag] = "Foo";

Object.defineProperty( Foo, Symbol.hasInstance, {
    value: function(inst) {
        return inst.greeting == "hello";
    }
} );

var a = new Foo( "hello" ),
    b = new Foo( "world" );

b[Symbol.toStringTag] = "cool";

a.toString();        // [object Foo]
String( b );         // [object cool]

a instanceof Foo;    // true
b instanceof Foo;    // false
```

原型（或实例本身）的 @@toStringTag 符号指定了在 [object ___] 字符串化时使用的字符串值。

@@hasInstance 符号是在构造器函数上的一个方法，接受实例对象值，通过返回 true 或 false 来指示这个值是否可以被认为是一个实例。

要在一个函数上设定 @@hasInstance，必须使用 Object.defineProperty(..)，因为 Function.prototype 上默认的那一个是 writable: false（不可写的）。参见本系列《你不知道的 JavaScript（上卷）》第二部分获取更多信息。

7.3.3 Symbol.species

在 3.4 节中，我们介绍了符号 @@species，这个符号控制要生成新实例时，类的内置方法使用哪一个构造器。

最常见的例子是，在创建 Array 的子类并想要定义继承的方法（比如 slice(..)）时使用哪一个构造器（是 Array(..) 还是自定义的子类）。默认情况下，调用 Array 子类实例上的 slice(..) 会创建这个子类的新实例，坦白说这很可能就是你想要的。

但是，你可以通过覆盖一个类的默认 @@species 定义来进行元编程：

```
class Cool {
    // 把@@species推迟到子类
    static get [Symbol.species]() { return this; }

    again() {
        return new this.constructor[Symbol.species]();
    }
}

class Fun extends Cool {}

class Awesome extends Cool {
    // 强制指定@@species为父构造器
    static get [Symbol.species]() { return Cool; }
}

var a = new Fun(),
    b = new Awesome(),
    c = a.again(),
    d = b.again();

c instanceof Fun;       // true
d instanceof Awesome;   // false
d instanceof Cool;      // true
```

就像前面代码中 Cool 的定义那样，内置原生构造器上 Symbol.species 的默认行为是 return this。在用户类上没有默认值，但是就像展示的那样，这个行为特性很容易模拟。

如果需要定义生成新实例的方法，使用 new this.constructor[Symbol.species](..) 模式元编程，而不要硬编码 new this.constructor(..) 或 new XYZ(..)。然后继承类就能够自定义 Symbol.species 来控制由哪个构造器产生这些实例。

7.3.4　Symbol.toPrimitive

在本系列《你不知道的 JavaScript（中卷)》第一部分中，我们讨论了抽象类型转换运算 `ToPrimitive`，它用在对象为了某个操作（比如比较 == 或者相加 +）必须被强制转换为一个原生类型值的时候。在 ES6 之前，没有办法控制这一行为。

而在 ES6 中，在任意对象值上作为属性的符号 `@@toPrimitivesymbol` 都可以通过指定一个方法来定制这个 ToPrimitive 强制转换。

考虑：

```
var arr = [1,2,3,4,5];

arr + 10;               // 1,2,3,4,510

arr[Symbol.toPrimitive] = function(hint) {
    if (hint == "default" || hint == "number") {
        // 求所有数字之和
        return this.reduce( function(acc,curr){
            return acc + curr;
        }, 0 );
    }
};

arr + 10;               // 25
```

`Symbol.toPrimitive` 方法根据调用 `ToPrimitive` 的运算期望的类型，会提供一个提示（hint）指定 `"string"`、`"number"` 或 `"default"`（这应该被解释为 `"number"`）。在前面的代码中，加法 + 运算没有提示（传入 `"default"`）。而乘法 * 运算提示为 `"number"`，`String(arr)` 提示为 `"string"`。

> 如果一个对象与另一个非对象值比较，== 运算符调用这个对象上的 `ToPrimitive` 方法时不指定提示——如果有 `@@toPrimitive` 方法的话，调用时提示为 `"default"`。但是，如果比较的两个值都是对象，== 的行为和 === 一样，也就是直接比较其引用。这种情况下完全不会调用 `@@toPrimitive`。参见本系列《你不知道的 JavaScript（中卷)》第一部分获取关于类型转换和抽象运算的更多信息。

7.3.5　正则表达式符号

对于正则表达式对象，有 4 个公开符号可以被覆盖，它们控制着这些正则表达式在 4 个对应的同名 `String.prototype` 函数中如何被使用。

- `@@match`：正则表达式的 `Symbol.match` 值是一个用于利用给定的正则表达式匹配一个字符串值的部分或全部内容的方法。如果传给 `String.prototype.match(..)` 一个正则表达式，那么用它来进行模式匹配。

ES6 规范的 21.2.5.6 节中给出了默认匹配算法（https://people.mozilla.org/~jorendorff/es6-draft.html#sec-regexp.prototype-@@match）。你可以覆盖这个默认算法并提供额外的正则匹配特性，比如向后断言（look-behind assertion）。

抽象运算 `isRegExp`（参见 6.5.4 节的“警示”内容）也使用了 `Symbol.match`，来确定对象是否会被用作正则表达式。要强制使得某个对象上的这个检查失败，使它不被当作正则表达式，可以把 `Symbol.match` 值设置为 `false`（或者任何为假的东西）。* `@@replace`：正则表达式的 `Symbol.replace` 值是一个方法，`String.prototype.replace(..)` 用它来替换一个字符串内匹配给定的正则表达式模式的一个或多个字符序列。

ES6 规范的 21.2.5.8 节中给出了替换的默认算法（https://people.mozilla.org/~jorendorff/es6-draft.html#sec-regexp.prototype-@@replace）。

覆盖默认算法的一个很棒的应用是提供额外的 `replacer` 参数选项，比如通过消耗 iterable 来产生连续替换值，支持 `"abaca".replace(/a/g,[1,2,3])` 产生 `"1b2c3"` 这样的形式。* `@@search`：正则表达式的 `Symbol.search` 值是一个方法，`String.prototype.search(..)` 用它来在另一个字符串中搜索一个匹配给定正则表达式的子串。

ES6 规范的 21.2.5.9 节中给出了搜索的默认算法（https://people.mozilla.org/~jorendorff/es6-draft.html#sec-regexp.prototype-@@search）。* `@@split`：正则表达式的 `Symbol.split` 值是一个方法，`String.prototype.split(..)` 用它把字符串在匹配给定正则表达式的分隔符处分割为子串。

ES6 规范的 21.2.5.11 节中给出了默认的分割算法（https://people.mozilla.org/~jorendorff/es6-draft.html#sec-regexp.prototype-@@split）。

如果你不够艺高人胆大的话，就不要覆盖内置正则表达式算法了！ JavaScript 的正则表达式引擎经过高度优化，所以你自己的用户代码很可能会慢上许多。这类元编程简洁强大，但是只应该在确实需要或能带来收益的时候才使用。

7.3.6 Symbol.isConcatSpreadable

符号 `@@isConcatSpreadable` 可以被定义为任意对象（比如数组或其他可迭代对象）的布尔型属性（`Symbol.isConcatSpreadable`），用来指示如果把它传给一个数组的 `concat(..)` 是否应该将其**展开**。

考虑：

```
var a = [1,2,3],
    b = [4,5,6];

b[Symbol.isConcatSpreadable] = false;

[].concat( a, b );      // [1,2,3,[4,5,6]]
```

7.3.7 Symbol.unscopables

符号 @@unscopables 可以被定义为任意对象的对象属性（Symbol.unscopables），用来指示使用 with 语句时哪些属性可以或不可以暴露为词法变量。

考虑：

```
var o = { a:1, b:2, c:3 },
    a = 10, b = 20, c = 30;

o[Symbol.unscopables] = {
    a: false,
    b: true,
    c: false
};

with (o) {
    console.log( a, b, c );      // 1 20 3
}
```

@@unscopables 对象中的 true 表示这个属性应该是 unscopable 的，因此会从词法作用域变量中被过滤出去。false 表示可以将其包含到词法作用域变量中。

> strict 模式下不允许 with 语句，因此应当被认为是语言的过时特性。不要使用它。参见本系列《你不知道的 JavaScript（上卷）》第一部分获取更多信息。因为应该避免使用 with，所以符号 @@unscopables 也没有太大意义。

7.4 代理

ES6 中新增的最明显的元编程特性之一是 Proxy（代理）特性。

代理是一种由你创建的特殊的对象，它“封装”另一个普通对象——或者说挡在这个普通对象的前面。你可以在代理对象上注册特殊的处理函数（也就是 trap），代理上执行各种操作的时候会调用这个程序。这些处理函数除了把操作转发给原始目标 / 被封装对象之外，还有机会执行额外的逻辑。

你可以在代理上定义的 trap 处理函数的一个例子是 get，当你试图访问对象属性的时候，它拦截 [[Get]] 运算。考虑：

```
var obj = { a: 1 },
    handlers = {
        get(target,key,context) {
            // 注意：target === obj,
            // context === pobj
            console.log( "accessing: ", key );
            return Reflect.get(
```

```
                target, key, context
            );
        }
    },
    pobj = new Proxy( obj, handlers );

obj.a;
// 1

pobj.a;
// accessing: a
// 1
```

我们在 handlers（Proxy(..) 的第二个参数）对象上声明了一个 get(..) 处理函数命名方法，它接受一个 target 对象的引用（obj）、key 属性名 ("a") **粗体文字**以及 self/ 接收者 / 代理（pobj）。

在跟踪语句 console.log(..) 之后，我们把对 obj 的操作通过 Reflect.get(..)“转发”。下一小节中会介绍 APIReflect，这里只要了解每个可用的代理 trap 都有一个对应的同名 Reflect 函数即可。

这里的映射是有意对称的。每个代理处理函数在对应的元编程任务执行的时候进行拦截，而每个 Reflect 工具在一个对象上执行相应的元编程任务。每个代理处理函数都有一个自动调用相应的 Reflect 工具的默认定义。几乎可以确定 Proxy 和 Reflect 总是这么协同工作的。

下面所列出的是在目标对象 / 函数代理上可以定义的处理函数，以及它们如何 / 何时被触发。

get(..)

通过 [[Get]]，在代理上访问一个属性（Reflect.get(..)、. 属性运算符或 [..] 属性运算符）

set(..)

通过 [[Set]]，在代理上设置一个属性值（Reflect.set(..)、赋值运算符 = 或目标为对象属性的解构赋值）

deleteProperty(..)

通过 [[Delete]]，从代理对象上删除一个属性（Reflect.deleteProperty(..) 或 delete）

apply(..)（**如果目标为函数**）

通过 [[Call]]，将代理作为普通函数 / 方法调用（Reflect.apply(..)、call(..)、apply(..) 或 (..) 调用运算符）

construct(..)（**如果目标为构造函数**）

通过 [[Construct]]，将代理作为构造函数调用（Reflect.construct(..) 或 new）

getOwnPropertyDescriptor(..)

通过 [[GetOwnProperty]]，从代理中提取一个属性描述符（Object.getOwnPropertyDescriptor(..) 或 Reflect.getOwnPropertyDescriptor(..)）

defineProperty(..)

通过 [[DefineOwnProperty]]，在代理上设置一个属性描述符（Object.defineProperty(..) 或 Reflect.defineProperty(..)）

getPrototypeOf(..)

通过 [[GetPrototypeOf]]，得到代理的 [[Prototype]]（Object.getPrototypeOf(..)、Reflect.getPrototypeOf(..)、__proto__、Object#isPrototypeOf(..) 或 instanceof）

setPrototypeOf(..)

通过 [[SetPrototypeOf]]，设置代理的 [[Prototype]]（Object.setPrototypeOf(..)、Reflect.setPrototypeOf(..) 或 __proto__）

preventExtensions(..)

通过 [[PreventExtensions]]，使得代理变成不可扩展的（Object.prevent Extensions(..) 或 Reflect.preventExtensions(..)）

isExtensible(..)

通过 [[IsExtensible]]，检测代理是否可扩展（Object.isExtensible(..) 或 Reflect.isExtensible(..)）

ownKeys(..)

通过 [[OwnPropertyKeys]]，提取代理自己的属性和 / 或符号属性（Object.keys(..)、Object.getOwnPropertyNames(..)、Object.getOwnSymbolProperties(..)、Reflect.ownKeys(..) 或 JSON.stringify(..)）

enumerate(..)

通过 [[Enumerate]]，取得代理拥有的和“继承来的”可枚举属性的迭代器（Reflect.enumerate(..) 或 for..in）

has(..)

通过 [[HasProperty]]，检查代理是否拥有或者“继承了”某个属性（Reflect.has(..)、Object#hasOwnProperty(..) 或 "prop" in obj）

要了解这里每个元编程任务的更多信息，参见 7.5 节。

除了上面列出的会触发各种 trap 的动作，某些 trap 是由其他 trap 的默认动作间接触发的。比如：

```
var handlers = {
        getOwnPropertyDescriptor(target,prop) {
            console.log(
                "getOwnPropertyDescriptor"
            );
            return Object.getOwnPropertyDescriptor(
                target, prop
            );
        },
        defineProperty(target,prop,desc){
            console.log( "defineProperty" );
            return Object.defineProperty(
                target, prop, desc
            );
        }
    },
    proxy = new Proxy( {}, handlers );

proxy.a = 2;
// getOwnPropertyDescriptor
// defineProperty
```

`getOwnPropertyDescriptor(..)` 和 `defineProperty(..)` 处理函数是在设定属性值（不管是新增的还是更新已有的）时由默认 `set(..)` 处理函数的步骤触发的。如果你也自定义了 `set(..)` 处理函数，那么在 `context`（不是 `target`！）上可以（也可以不）进行相应的调用，这些调用会触发这些代理 trap。

7.4.1 代理局限性

可以在对象上执行的很广泛的一组基本操作都可以通过这些元编程处理函数 trap。但有一些操作是无法（至少现在）拦截的。

比如，下面这些操作都不会 trap 并从代理 `pobj` 转发到目标 `obj`：

```
var obj = { a:1, b:2 },
    handlers = { .. },
    pobj = new Proxy( obj, handlers );

typeof obj;
String( obj );
```

```
obj + "";
obj == pobj;
obj === pobj
```

也许在未来，语言中的这些底层基本运算可能会有更多变成可拦截的，那将为我们从内部扩展 JavaScript 提供更强大的能力。

> 代理处理函数总会有一些不变性（invariant），亦即不能被覆盖的行为。比如，`isExtensible(..)` 处理函数的返回值总会被类型转换为 `boolean`。这些不变性限制了自定义代理行为的能力，但它们的目的只是为了防止你创建诡异或罕见（或者不一致）的行为。这些不变性条件非常复杂，所以这里我们不会完整介绍，这篇文章（http://www.2ality.com/2014/12/es6-proxies.html#invariants）对此给出了很棒的介绍。

7.4.2　可取消代理

普通代理总是陷入到目标对象，并且在创建之后不能修改——只要还保持着对这个代理的引用，代理的机制就将维持下去。但是，可能会存在这样的情况，比如你想要创建一个在你想要停止它作为代理时便可以被停用的代理。解决方案是创建**可取消代理**（revocable proxy）：

```
var obj = { a: 1 },
    handlers = {
        get(target,key,context) {
            // 注意：target === obj,
            // context === pobj
            console.log( "accessing: ", key );
            return target[key];
        }
    },
    { proxy: pobj, revoke: prevoke } =
        Proxy.revocable( obj, handlers );

pobj.a;
// accessing: a
// 1

// 然后：
prevoke();

pobj.a;
// TypeError
```

可取消代理用 `Proxy.revocable(..)` 创建，这是一个普通函数，而不像 `Proxy(..)` 一样是构造器。除此之外，它接收同样的两个参数：target 和 handlers。

和 `new Proxy(..)` 不一样，`Proxy.revocable(..)` 的返回值不是代理本身。而是一个有两个属性——proxy 和 revode 的对象，我们使用对象解构（参见 2.4 节）把这两个属性分别赋

给变量 `pobj` 和 `prevoke()`。

一旦可取消代理被取消，任何对它的访问（触发它的任意 trap）都会抛出 `TypeError`。

可取消代理的一个可能应用场景是，在你的应用中把代理分发到第三方，其中管理你的模型数据，而不是给出真实模型本身的引用。如果你的模型对象改变或者被替换，就可以使分发出去的代理失效，这样第三方能够（通过错误！）知晓变化并请求更新到这个模型的引用。

7.4.3　使用代理

这些处理函数为元编程带来的好处是显而易见的。我们可以拦截（并覆盖）对象的几乎所有行为，这意味着我们可以以强有力的方式扩展对象特性超出核心 JavaScript 内容。这里将会通过几种模式实例来探索这些可能性。

1. 代理在先，代理在后

我们在前面介绍过，通常可以把代理看作是对目标对象的“包装”。在这种意义上，代理成为了代码交互的主要对象，而实际目标对象保持隐藏 / 被保护的状态。

你可能这么做是因为你想要把对象传入到某个无法被完全“信任”的环境，因此需要为对它的访问增强规范性，而不是把对象本身传入。

考虑：

```
var messages = [],
    handlers = {
        get(target,key) {
            // 字符串值？
            if (typeof target[key] == "string") {
                // 过滤掉标点符号
                return target[key]
                    .replace( /[^\w]/g, "" );
            }

            // 所有其他的传递下去
            return target[key];
        },
        set(target,key,val) {
            // 设定唯一字符串，改为小写
            if (typeof val == "string") {
                val = val.toLowerCase();
                if (target.indexOf( val ) == -1) {
                    target.push(
                        val.toLowerCase()
                    );
                }
            }
            return true;
```

```
            }
        },
        messages_proxy =
            new Proxy( messages, handlers );

    // 其他某处：
    messages_proxy.push(
        "heLLo...", 42, "wOrlD!!", "WoRld!!"
    );

    messages_proxy.forEach( function(val){
        console.log(val);
    } );
    // hello world

    messages.forEach( function(val){
        console.log(val);
    } );
    // hello... world!!
```

我称之为**代理在先**（proxy first）设计，因为我们首先（主要、完全）与代理交互。

通过与 `messages_proxy` 交互来增加某些特殊的规则，这些是 `messages` 本身没有的。我们只在值为字符串并且是唯一值的时候才添加这个元素；我们还将这个值变为小写。在从 `messages_proxy` 提取值的时候，我们过滤掉了字符串中的所有标点符号。

另外，我们也可以完全反转这个模式，让目标与代理交流，而不是代理与目标交流。这样，代码只能与主对象交互。这个回退方式的最简单实现就是把 proxy 对象放到主对象的 `[[Prototype]]` 链中。

考虑：

```
    var handlers = {
            get(target,key,context) {
                return function() {
                    context.speak(key + "!");
                };
            }
        },
        catchall = new Proxy( {}, handlers ),
        greeter = {
            speak(who = "someone") {
                console.log( "hello", who );
            }
        };

    // 设定greeter回退到catchall
    Object.setPrototypeOf( greeter, catchall );

    greeter.speak();                  // hello someone
    greeter.speak( "world" );         // hello world

    greeter.everyone();               // hello everyone!
```

这里直接与 `greeter` 而不是 `catchall` 交流。当我们调用 `speak(..)` 的时候，它在 `greeter` 上被找到并直接使用。但是当我们试图访问像 `everyone()` 这样的方法的时候，这个函数在 `greeter` 上并不存在。

默认的对象属性行为是检查 `[[Prototype]]` 链（参见本系列《你不知道的 JavaScript（上卷）》第二部分），所以会查看 `catchall` 是否有 `everyone` 属性。然后代理的 `get()` 处理函数介入并返回一个用访问的属性名（`"everyone"`）调用 `speak(..)` 的函数。

我把这个模式称为**代理在后**（proxy last），因为在这里代理只作为最后的保障。

2. "No Such Property/Method"

有一个关于 JavaScript 的常见抱怨，在你试着访问或设置一个还不存在的属性时，默认情况下对象不是非常具有防御性。你可能希望预先定义好一个对象的所有属性 / 方法之后，访问不存在的属性名时能够抛出一个错误。

我们可以通过代理实现这一点，代理在先或代理在后设计都可以。两种情况我们都考虑一下：

```
var obj = {
        a: 1,
        foo() {
            console.log( "a:", this.a );
        }
    },
    handlers = {
        get(target,key,context) {
            if (Reflect.has( target, key )) {
                return Reflect.get(
                    target, key, context
                );
            }
            else {
                throw "No such property/method!";
            }
        },
        set(target,key,val,context) {
            if (Reflect.has( target, key )) {
                return Reflect.set(
                    target, key, val, context
                );
            }
            else {
                throw "No such property/method!";
            }
        }
    },
    pobj = new Proxy( obj, handlers );

pobj.a = 3;
pobj.foo();      // a: 3
```

```
pobj.b = 4;      // Error: No such property/method!
pobj.bar();      // Error: No such property/method!
```

对于 get(..) 和 set(..)，我们都只在目标对象的属性存在的时候才转发这个操作；否则抛出错误。主对象代码应该与代理对象（pobj）交流，因为它截获这些动作以提供保护。

现在，考虑转换为代理在后设计：

```
var handlers = {
        get() {
            throw "No such property/method!";
        },
        set() {
            throw "No such property/method!";
        }
    },
    pobj = new Proxy( {}, handlers ),
    obj = {
        a: 1,
        foo() {
            console.log( "a:", this.a );
        }
    };

// 设定obj回退到pobj
Object.setPrototypeOf( obj, pobj );

obj.a = 3;
obj.foo();        // a: 3

obj.b = 4;        // Error: No such property/method!
obj.bar();        // Error: No such property/method!
```

考虑到处理函数的定义方式，这里的代理在后设计更简单一些。与截获 [[Get]] 和 [[Set]] 操作并且只在目标属性存在情况下才转发不同，我们依赖于这样一个事实：如果 [[Get]] 或 [[Set]] 进入我们的 pobj 回退，此时这个动作已经遍历了整个 [[Prototype]] 链并且没有发现匹配的属性。这时候我们可以自由抛出错误。不错吧？

3. 代理 hack [[Prototype]] 链

[[Prototype]] 机制运作的主要通道是 [[Get]] 运算。当直接对象中没有找到一个属性的时候，[[Get]] 会自动把这个运算转给 [[Prototype]] 对象处理。

这意味着你可以使用代理的 get(..) trap 来模拟或扩展这个 [[Prototype]] 机制的概念。

我们将考虑的第一个 hack 就是创建两个对象，通过 [[Prototype]] 连成环状（或者，至少看起来是这样！）。实际上并不能创建一个真正的 [[Prototype]] 环，因为引擎会抛出错误。但是可以用代理模拟！

考虑：

```
var handlers = {
        get(target,key,context) {
            if (Reflect.has( target, key )) {
                return Reflect.get(
                    target, key, context
                );
            }
            // 伪环状[[Prototype]]
            else {
                return Reflect.get(
                    target[
                        Symbol.for( "[[Prototype]]" )
                    ],
                    key,
                    context
                );
            }
        }
    },
    obj1 = new Proxy(
        {
            name: "obj-1",
            foo() {
                console.log( "foo:", this.name );
            }
        },
        handlers ),
    },
    obj2 = Object.assign(
        Object.create( obj1 ),
        {
            name: "obj-2",
            bar() {
                console.log( "bar:", this.name );
                this.foo();
            }
        }
    );

// 伪环状[[Prototype]]链接
obj1[ Symbol.for( "[[Prototype]]" ) ] = obj2;

obj1.bar();
// bar: obj-1 <-- 通过代理伪装[[Prototype]]
// foo: obj-1 <-- this上下文依然保留着

obj2.foo();
// foo: obj-2 <-- 通过[[Prototype]]
```

在这个例子中，我们不需要代理 / 转发 `[[Set]]`，所以比较简单。要完整模拟 `[[Prototype]]`，需要实现一个 `set(..)` 处理函数来搜索 `[[Prototype]]` 链寻找匹配的属性，并遵守其描述符特性（比如 set，可写的）。参见本系列《你不知道的 JavaScript（上卷）》第二部分。

在前面的代码中，通过 Object.create(..) 语句 obj2 [[Prototype]] 链接到了 obj1。而为了创建反向（环）的链接，我们在 obj1 符号位置 Symbol.for("[[Prototype]]")（参见 2.13 节）处创建了属性。这个符号可能看起来有点特殊 / 神奇，但实际上并非如此。它只是给我提供了一个方便的与我正在执行的任务关联的命名钩子，以便语义上引用。

然后，代理的 get(..) 处理函数首先查看这个代理上是否有请求的 key。如果没有，就手动把这个运算转发给保存在 target 的 Symbol.for("[[Prototype]]") 位置中的对象引用。

这种模式的一个重要优点是，obj1 和 obj2 的定义几乎不会受到在它们之间建立的这种环状关系的影响。尽管为了简洁的缘故，前面代码把所有的代码都纠缠到了一起，但是仔细观察可以看到，代理处理函数的逻辑完全是通用的（并不具体了解 obj1 和 obj2 的细节）。所以，这段逻辑可提取出来封装为一个单独的辅助函数，比如 setCircularPrototypeOf(..)。我们把这个实现留给读者作为练习。

既然已经了解了如何通过 get(..) 来模拟一个 [[Prototype]] 链接，现在让我们来深入 hack 一下。不用环状 [[Prototype]]，用多个 [[Prototype]] 链接（也就是“多继承”）怎么样？实际上这非常简单直接：

```
var obj1 = {
        name: "obj-1",
        foo() {
            console.log( "obj1.foo:", this.name );
        },
    },
    obj2 = {
        name: "obj-2",
        foo() {
            console.log( "obj2.foo:", this.name );
        },
        bar() {
            console.log( "obj2.bar:", this.name );
        }
    },
    handlers = {
        get(target,key,context) {
            if (Reflect.has( target, key )) {
                return Reflect.get(
                    target, key, context
                );
            }
            // 伪装多个[[Prototype]]
            else {
                for (var P of target[
                    Symbol.for( "[[Prototype]]" )
                ]) {
                    if (Reflect.has( P, key )) {
                        return Reflect.get(
                            P, key, context
                        );
```

```
                    }
                }
            }
        }
    },
    obj3 = new Proxy(
        {
            name: "obj-3",
            baz() {
                this.foo();
                this.bar();
            }
        },
        handlers
    );

// 伪装多个[[Prototype]]链接
obj3[ Symbol.for( "[[Prototype]]" ) ] = [
    obj1, obj2
];

obj3.baz();
// obj1.foo: obj-3
// obj2.bar: obj-3
```

> 正如前面环状 [[Prototype]] 例子之后的注释中提到的一样，我们没有实现 set(..) 处理函数，但是要实现一个完整解决方案，模拟 [[Set]] 动作作为普通的 [[Prototype]] 行为是必要的。

obj3 建立了多委托到 obj1 和 obj2。在 obj3.baz() 中，this.foo() 调用最后从 obj1 中提出 foo()（先到先得，虽然 obj2 上也有一个 foo()）。如果我们把链接重新排序为 obj2、obj1，就会找到并使用 obj2.foo()。

而现在 this.bar() 调用不会在 obj1 上找到 bar()，所以它会陷入检查 obj2，在其中找到匹配。

obj1 和 obj2 表示 obj3 的两条平行的 [[Prototype]] 链。obj1 和 / 或 obj2 本身也可以有普通的 [[Prototype]] 委托到其他对象，或者本身也可以是一个多委托的代理（就像 obj3 一样）。

就像前面的环状 [[Prototype]] 链例子一样，obj1、obj2 和 obj3 的定义与通用的处理多委托代理的逻辑几乎是完全分离的。要定义一个像 setPrototypesOf(..)（注意这个表示复数的“s”）这样的工具接收一个主对象和一个对象列表来模拟多 [[Prototype]] 链接是很简单的。我们还是把这个实现留给读者作为练习。

希望在各种各样的例子之后代理的威力现在变得明朗了。代理使得很多其他威力强大的元编程任务成为可能。

7.5 Reflect API

Reflect 对象是一个平凡对象（就像 Math），不像其他内置原生值一样是函数 / 构造器。

它持有对应于各种可控的元编程任务的静态函数。这些函数一对一对应着代理可以定义的处理函数方法（trap）。

这些函数中的一部分看起来和 Object 上的同名函数类似：

- Reflect.getOwnPropertyDescriptor(..);
- Reflect.defineProperty(..);
- Reflect.getPrototypeOf(..);
- Reflect.setPrototypeOf(..);
- Reflect.preventExtensions(..);
- Reflect.isExtensible(..)。

一般来说这些工具和 Object.* 的对应工具行为方式类似。但是，有一个区别是如果第一个参数（目标对象）不是对象的话，Object.* 相应工具会试图把它类型转换为一个对象。而这种情况下 Reflect.* 方法只会抛出一个错误。

可以使用下面这些工具访问 / 查看一个对象键：

Reflect.ownKeys(..)

返回所有“拥有”的（不是“继承”的）键的列表，就像 Object.getOwnPropertyNames(..) 和 Object.getOwnPropertySymbols(..) 返回的一样。关于键的顺序参见后面的“属性排序”一节。

Reflect.enumerate(..)

返回一个产生所有（拥有的和“继承的”）**可枚举的**（enumerable）非符号键集合的迭代器（参见本系列《你不知道的 JavaScript（上卷）》第二部分）。本质上说，这个键的集合和 foo..in 循环处理的那个键的集合是一样的。关于键的顺序参见后面的“属性排序”一节。

Reflect.has(..)

实质上和 in 运算符一样，用于检查某个属性是否在某个对象上或者在它的 [[Prototype]] 链上。比如，Reflect.has(o, "foo") 实质上就是执行 "foo" in o。

函数调用和构造器调用可以通过使用下面这些工具手动执行，与普通的语法（比如，(..) 和 new）分开：

`Reflect.apply(..)`

举例来说，`Reflect.apply(foo,thisObj,[42,"bar"])` 以 `thisObj` 作为 `this` 调用 `foo(..)` 函数，传入参数 `42` 和 `"bar"`。

`Reflect.construct(..)`

举例来说，`Reflect.construct(foo,[42,"bar"])` 实质上就是调用 `new foo(42,"bar")`。

可以使用下面这些工具来手动执行对象属性访问、设置和删除。

`Reflect.get(..)`

举例来说，`Reflect.get(o,"foo")` 提取 `o.foo`。

`Reflect.set(..)`

举例来说，`Reflect.set(o,"foo",42)` 实质上就是执行 `o.foo = 42`。

`Reflect.deleteProperty(..)`

举例来说，`Reflect.deleteProperty(o,"foo")` 实质上就是执行 `delete o.foo`。

`Reflect` 的元编程能力提供了模拟各种语法特性的编程等价物，把之前隐藏的抽象操作暴露出来。比如，你可以利用这些能力扩展功能和 API，以实现**领域特定语言**（DSL）。

属性排序

在 ES6 之前，一个对象键 / 属性的列出顺序是依赖于具体实现，并未在规范中定义。一般来说，多数引擎按照创建的顺序进行枚举，虽然开发者们一直被强烈建议不要依赖于这个顺序。

对于 ES6 来说，拥有属性的列出顺序是由 `[[OwnPropertyKeys]]` 算法定义的（ES6 规范，9.1.12 节），这个算法产生所有拥有的属性（字符串或符号），不管是否可枚举。这个顺序只对 `Reflect.ownKeys(..)`（以及扩展的 `Object.getOwnPropertyNames(..)` 和 `Object.getOwnPropertySymbols(..)`）有保证。

其顺序为：

(1) 首先，按照数字上升排序，枚举所有整数索引拥有的属性；
(2) 然后，按照创建顺序枚举其余的拥有的字符串属性名；
(3) 最后，按照创建顺序枚举拥有的符号属性。

考虑：

```
var o = {};

o[Symbol("c")] = "yay";
```

```
o[2] = true;
o[1] = true;
o.b = "awesome";
o.a = "cool";

Reflect.ownKeys( o );               // [1,2,"b","a",Symbol(c)]
Object.getOwnPropertyNames( o );    // [1,2,"b","a"]
Object.getOwnPropertySymbols( o );  // [Symbol(c)]
```

另一方面，[[Enumerate]] 算法（ES6 规范，9.1.11 节）只从目标对象和它的 [[Prototype]] 链产生可枚举属性。它用于 Reflect.enumerate(..) 和 for..in。可以观察到的顺序和具体的实现相关，不由规范控制。

与之对比，Object.keys(..) 调用 [[OwnPropertyKeys]] 算法取得拥有的所有键的列表。但是，它会过滤掉不可枚举属性，然后把这个列表重新排序来遵循遗留的与实现相关的行为特性，特别是 JSON.stringify(..) 和 for..in。因此通过扩展，这个顺序也和 Reflect.enumerate(..) 顺序相匹配。

换句话说，所有这 4 种机制（Reflect.enumerate(..)、Object.keys(..)、for..in 和 JSON.stringify(..)）都会匹配同样的与具体实现相关的排序，尽管严格上说是通过不同的路径。

把这 4 种机制与 [[OwnPropertyKeys]] 的排序匹配的具体实现是允许的，但并不是必须的。尽管如此，你很可能会看到它们的排序特性是这样的：

```
var o = { a: 1, b: 2 };
var p = Object.create( o );
p.c = 3;
p.d = 4;

for (var prop of Reflect.enumerate( p )) {
    console.log( prop );
}
// c d a b

for (var prop in p) {
    console.log( prop );
}
// c d a b

JSON.stringify( p );
// {"c":3,"d":4}

Object.keys( p );
// ["c","d"]
```

总结一下：对于 ES6 来说，Reflect.ownKeys(..)、Object.getOwnPropertyNames(..) 和 Object.getOwnPropertySymbols(..) 的顺序都是可预测且可靠的，这由规范保证。所以依

赖于这个顺序的代码是安全的。

`Reflect.enumerate(..)`、`Object.keys(..)` 和 `for..in`（以及扩展的 `JSON.stringification(..)`）还像过去一样，可观察的顺序是相同的。但是这个顺序不再必须与 `Reflect.ownKeys(..)` 相同。在使用它们依赖于具体实现的顺序时仍然要小心。

7.6　特性测试

什么是特性测试？就是一种由你运行的用来判断一个特性是否可用的测试。有时候，这个测试不只是为了测试特性是否存在，还是为了测试特性是否符合指定的行为规范——特性可能存在但却是有问题的。

测试程序的运行环境，然后确定程序行为方式，这是一种元编程技术。

JavaScript 中最常用的特性测试是检查一个 API 是否存在，如果不存在的话，定义一个 polyfill（参见第 1 章）。比如：

```
if (!Number.isNaN) {
    Number.isNaN = function(x) {
        return x !== x;
    };
}
```

这段代码中的 `if` 语句是元编程：我们检查程序和它的运行环境，以此来确定是否应该继续以及如何继续。

但是如何测试涉及新语法的特性呢？

可能你会想到使用像下面这样的代码：

```
try {
    a = () => {};
    ARROW_FUNCS_ENABLED = true;
}
catch (err) {
    ARROW_FUNCS_ENABLED = false;
}
```

不幸的是，这行不通，因为我们的 JavaScript 程序是需要编译的。所以，如果引擎还不支持 ES6 箭头函数的话，就会停在 `() => {}` 语法处。这样你程序中的一个语法错误会使其无法运行，你的程序也就无法根据特性是否被支持而作出不同的反应。

要针对语法相关的特性进行特性测试的元编程，我们需要一种方法能够把测试与程序的初始编译步骤隔绝开来。比如，如果我们可以把测试代码放在一个字符串里，那么 JavaScript 引擎默认就不会试图编译这个字符串的内容，直到要求它这么做。

我们直接跳到使用 `eval(..)` 怎么样？

还没那么快，参见本系列《你不知道的 JavaScript（上卷）》第一部分，其中介绍了为什么 `eval(..)` 不是一个好主意。但是还有一种缺点少一些的选择：`Function(..)` 构造器。

考虑：

```
try {
    new Function( "( () => {} )" );
    ARROW_FUNCS_ENABLED = true;
}
catch (err) {
    ARROW_FUNCS_ENABLED = false;
}
```

好吧，现在我们就是在通过元编程确定像箭头函数这样的特性是否能在当前引擎上编译。你可能会想知道，拿到了这个信息又能做什么呢？

有了 API 的存在性检查，并且定义了用作退路的 API polyfill，那么测试成功和失败后的路径是很清晰的。但是知道了 `ARROW_FUNCS_ENABLED` 为 `true` 还是 `false` 这个信息之后又能做什么呢？

引擎不支持的语法特性不能出现在一个文件中，因此，不能在这个文件中分别使用或是不使用这个语法定义不同的函数。

你能做的是，使用这个测试来确定应该加载一组 JavaScript 文件中的哪一个。举例来说，如果在你的 JavaScript 应用程序的引导程序（bootstrapper）中有一组这样的特性测试，就可以通过测试环境来确定你的 ES6 代码是能够直接加载运行，还是需要加载代码的 transpile 版本（参见第 1 章）。

这种技术叫作**分批发布**（split delivery）。

事实表明，有时候你的 ES6 JavaScript 程序能够完整“原生”运行在 ES6+ 浏览器中，但有时候又需要 transpilation 运行在前 ES6 浏览器中。如果总是加载使用 transpile 的代码，甚至在新的 ES6 兼容环境中也是这样，那么至少有时候是在运行非优化的代码。这并不理想。

分批发布更复杂也更高级，但它是一种更成熟、更健壮的方法，可以弥合代码编写和程序运行浏览器支持的特性之间的裂隙。

FeatureTests.io

为所有 ES6+ 语法和语义行为特性定义特性测试，可能是你并不想亲自动手的令人望而却步的工作。因为这些测试需要动态编译（`new Function(..)`），所以会有一些不幸的性能损失。

另外，每次程序运行的时候都要运行这些测试可能也是一种浪费，因为一般用户浏览器最多也只是几个星期才更新一次，即便如此，也不是每次更新都会出现新特性。

最后，管理应用到具体代码的特性测试列表也是容易失控和出错的——你的程序几乎不会使用所有 ES6 特性。

FeatureTests.io（https://featuretests.io）为此提供了解决方案。

你可以在自己的页面中加载这个服务库，然后它会加载最新的测试定义并运行所有特性测试。如果可能的话，就使用 Web Worker 的后台进程实现这些，以减少性能损失。它还会使用 LocalStorage 持久存储缓存结果，并且其实现方式使得你访问的所有使用这个服务的站点都可以共享缓存，这显著降低了每个浏览器实例上运行这些测试的所需频率。

这样你得到了每个用户浏览器的运行时特性测试，然后就可以巧妙使用这些测试结果根据用户的环境为用户提供最合适的代码（不多也不少）。

另外，这个服务提供了工具和 API 来扫描你的文件以确定需要哪些特性，所以你可以完全自动化分批发布构建过程。

FeatureTests.io 使得应用所有 ES6 及之后的特性测试，确保在给定的环境中只加载运行最优代码成为可能。

7.7　尾递归调用（Tail Call Optimization,TCO）

通常，在一个函数内部调用另一个函数的时候，会分配第二个**栈帧**来独立管理第二个函数调用的变量 / 状态。这个分配不但消耗处理时间，也消耗了额外的内存。

通常调用栈链最多有 10~15 个从一个函数到另一个函数的跳转。这种情况下，内存使用并不会造成任何实际问题。

但是，当考虑到递归编程的时候（一个函数重复调用自身）——或者两个或多个函数彼此调用形成递归——调用栈的深度很容易达到成百上千，甚至更多。如果内存的使用无限制地增长下去，你可能看到了它将导致的问题。

JavaScript 引擎不得不设置一个武断的限制来防止这种编程技术引起浏览器和设备内存耗尽而崩溃。这也是为什么达到这个限制的时候我们会得到烦人的“RangeError: Maximum call stack size exceeded”。

> 调用栈深度限制不由规范控制。它是依赖于具体实现的，并且根据浏览器和设备不同而有所不同。在编码的时候不要对观察到的具体限制值有任何强假定，因为它很可能根据发布版本的不同而有所不同。

有一些称为**尾调用**（tail call）的函数调用模式，可以以避免额外栈帧分配的方式进行优化。如果可以避免额外的分配，就没有理由任意限制调用栈深度，所以引擎就可以不设置这个限制。

尾调用是一个 `return` 函数调用的语句，除了调用后返回其返回值之外没有任何其他动作。

这个优化只在 `strict` 模式下应用。这又是一个要坚持编写 `strict` 模式代码的原因！

下面是一个不在尾位置的函数调用：

```
"use strict";

function foo(x) {
    return x * 2;
}

function bar(x) {
    // 这不是尾调用
    return 1 + foo( x );
}

bar( 10 );              // 21
```

`foo(x)` 调用完毕后还得执行 `1 + ..`，所以 `bar(..)` 调用的状态需要被保留。

但下面代码展示的对 `foo(..)` 和 `bar(..)` 的调用都处于尾位置，因为它们是在其代码路径上发生的最后一件事（除了 `return`）：

```
"use strict";

function foo(x) {
    return x * 2;
}

function bar(x) {
    x = x + 1;
    if (x > 10) {
        return foo( x );
    }
    else {
        return bar( x + 1 );
    }
}

bar( 5 );               // 24
bar( 15 );              // 32
```

在这个程序中，`bar(..)` 显然是递归，而 `foo(..)` 只是一个普通函数调用。在这两种情况下，函数调用都处于**合适的尾位置**（proper tail position）。`x + 1` 在 `bar(..)` 调用之前求值，在调用结束后，所做的只有 `return`。

这些形式的正确尾调用（Proper Tail Call，PTC）是可以被优化的——称为尾调用优化（Tail Call Optimization，TCO）——于是额外的栈帧分配是不需要的。引擎不需要对下一个函数调用创建一个新的栈帧，只需复用已有的栈帧。这能够工作是因为一个函数不需要保留任何当前状态——在 PTC 之后不需要这个状态做任何事情。

TCO 意味着对调用栈的允许深度没有任何限度。对于一般程序中的普通函数调用，这个技巧有些许优化，但更重要的是打开了在程序表达中使用递归的大门，甚至是调用栈的调用深度可能达到成千上万的时候。

现在我们不再只把递归作为解决问题的理论方案了，而是可以实际将其用在 JavaScript 程序中！

对于 ES6 来说，不管是否为递归，所有的 PTC 都应该以这种方式优化。

7.7.1 尾调用重写

但这里的问题是只有 PTC 可以被优化；非 PTC 当然仍然可以工作，但会像以前一样触发栈帧分配。如果你希望这个优化介入的话，需要认真设计函数结构支持 PTC。

如果有一个函数不是以 PTC 方式编写的，那么你可能会需要手动重新安排代码以适合 TCO。

考虑：

```
"use strict";

function foo(x) {
    if (x <= 1) return 1;
    return (x / 2) + foo( x - 1 );
}

foo( 123456 );          // RangeError
```

调用 `foo(x-1)` 不是 PTC，因为它的结果每次在 `return` 之前要加上 `(x / 2)`。

但是，要想使这段代码适合 ES6 引擎 TCO，可以这样重写：

```
"use strict";

var foo = (function(){
    function _foo(acc,x) {
        if (x <= 1) return acc;
        return _foo( (x / 2) + acc, x - 1 );
    }

    return function(x) {
        return _foo( 1, x );
```

```
    };
})();

foo( 123456 );          // 3810376848.5
```

如果你在实现了 TCO 的 ES6 引擎中运行前面的代码，会得到如前显示的 3810376848.5。然而，它在非 TCO 引擎里仍然会因 RangeError 而失败。

7.7.2 非 TCO 优化

还有几种其他技术可以用来重写代码，使得不需要每次调用时都增长栈。

其中一种这样的技术叫作 trampolining，它相当于把每个部分结果用一个函数表示，这些函数或者返回另外一个部分结果函数，或者返回最终结果。然后就只需要循环直到得到的结果不是函数，得到的就是最终结果。

考虑：

```
"use strict";

function trampoline( res ) {
    while (typeof res == "function") {
        res = res();
    }
    return res;
}

var foo = (function(){
    function _foo(acc,x) {
        if (x <= 1) return acc;
        return function partial(){
            return _foo( (x / 2) + acc, x - 1 );
        };
    }

    return function(x) {
        return trampoline( _foo( 1, x ) );
    };
})();

foo( 123456 );          // 3810376848.5
```

这个重写需要最小的改动来把递归转化为 trampoline(..) 中的循环。

(1) 首先，把 return _foo .. 一行封装在 return partial() { .. 函数表达式中。
(2) 然后，把 _foo(1,x) 调用封装在 trampoline(..) 调用中。

这个技术不限制调用栈的原因是，每个内部的 partial(..) 函数只是返回到 trampoline(..) 的 while 循环中，trampolining 运行函数并进行下一次的循环迭代。换句话

说，partial(..) 不会递归调用自身，它只是返回另一个函数。栈深度保持不变，所以可以运行任意长的时间。

通过这种方式实现的 trampolining 使用了内层 partial() 函数在变量 x 和 acc 上的闭包，在迭代之间保持状态。其优点是把循环逻辑抽出到了可复用的 trampoline(..) 工具函数，很多库都提供了它的各种版本。可以在你的程序中用不同的 trampoline 算法多次复用 trampoline(..)。

当然，如果真的需要深度优化（不需考虑可复用性），那么可以丢弃闭包状态，用一个循环把 acc 信息的状态追踪在线化放在一个函数作用域内。这种技术一般称为**递归展开**：

```
"use strict";

function foo(x) {
    var acc = 1;
    while (x > 1) {
        acc = (x / 2) + acc;
        x = x - 1;
    }
    return acc;
}

foo( 123456 );          // 3810376848.5
```

算法的这种表达方式可读性更高，很可能也是我们前面探索的各种形式中性能（严格说来）最高的。所以这个方案显然是最好的，你可能会奇怪为什么还要用其他方法。

下面是两个使我们并不想总是手动展开递归的原因。

- 这里没有为了可复用性把 trampolining（循环）逻辑提取出来，而是将它在线化了。在只需要考虑一个例子的时候这还合适，但如果你的程序中有多个这种情况，很可能需要提高复用度来保持代码更简短、更易管理。
- 这里的例子非常简单，只是用来展示各种不同的形式。但在实践中，递归算法中还有很多更复杂的逻辑，比如互相递归（不只一个函数调用自身）。

对这个无底洞探索越深，就会发现展开优化变得越手动化和错综复杂。你很快就会丧失所有前面得到的可读性价值。递归的最主要优势，即使是 PTC 形式，就是它保留了算法的可读性，并将性能优化的担子扔给引擎。

如果以 PTC 的形式编写算法，ES6 引擎就会应用 TCO，代码就会以常数栈深度（通过重用栈帧）运行。你在得到递归的可读性的同时，也得到了几乎没有损失的性能以及不受限制的运行长度。

7.7.3 元在何处

TCO 和元编程又有什么关系呢？

正如我们在 7.6 节中介绍的，可以在运行时判断引擎支持哪些特性。其中就包括 TCO，尽管确定方法是十分暴力的。考虑：

```
"use strict";

try {
    (function foo(x){
        if (x < 5E5) return foo( x + 1 );
    })( 1 );

    TCO_ENABLED = true;
}
catch (err) {
    TCO_ENABLED = false;
}
```

在非 TCO 引擎中，递归循环最终会失败，抛出一个异常被 `try..catch` 捕获。换句话说，有了 TCO，循环才能完成。

不怎么样，对吧？

而围绕着 TCO 特性（或者，这个特性的缺失）的元编程对我们的代码有什么好处呢？简单的答案是，可以通过这种特性测试来决定是加载使用递归的应用代码版本，还是转换 / transpile 为不需要递归的版本。

自适应代码

还有另外一种看问题的方式：

```
"use strict";

function foo(x) {
    function _foo() {
        if (x > 1) {
            acc = acc + (x / 2);
            x = x - 1;
            return _foo();
        }
    }

    var acc = 1;

    while (x > 1) {
        try {
            _foo();
        }
        catch (err) { }
```

```
    }

    return acc;
}

foo( 123456 );          // 3810376848.5
```

这个算法尽可能多地使用了递归，但是是通过作用域内变量 x 和 acc 保持进展状态。如果整个问题都可以不出错地通过递归解决，那么很好。如果引擎在某处杀死了递归，我们就会在 try..catch 中捕获到，然后再试一次，继续我们其余的工作。

我把这种形式看作是一种元编程，理由是在运行时探索引擎的能力来（递归地）完成任务，并且为可能的（非 TCO）引擎局限提供了替代版本。

第一眼（甚至第二眼！）看上去，我敢说比起前面的几个版本，你会觉得这段代码要丑陋许多。运行起来它也要慢得多（在非 TCO 环境中大量运行的情况下）。

这个对递归栈限制的“解决方案”的主要优点，除了即使在非 TCO 引擎中也能够完成任意规模的任务之外，就是比前面展示的 trampolining 技术和手动展开技术更灵活。

本质上说，这个例子中的 _foo() 可以替换为几乎任意递归任务，即使是互相递归。其余部分是任何算法都适用的样板。

唯一的“catch”是为了能够在递归达到限制的时候恢复，递归的状态必须要用递归函数之外的作用域变量维护。我们通过把 x 和 acc 放在 _foo() 函数之外，而不是像之前一样把它们作为参数传递给 _foo() 来实现这一点。

几乎所有递归算法都可以被改造为用这种方式工作。这意味着这是在你的程序中进行最小重写就能利用 TCO 递归的最广泛的可行方法。

这种方法也使用了 PTC，意味着从旧有浏览器中使用多次循环（递归批处理）运行到 ES6+ 环境中充分利用 TCO 的递归，这段代码将会得到显著提高。我认为这很酷！

7.8 小结

元编程是指把程序的逻辑转向关注自身（或自身的运行时环境），要么是为了查看自己的结构，要么是为了修改它。元编程的主要价值是扩展语言的一般机制来提供额外的新功能。

在 ES6 之前，JavaScript 已经有了不少的元编程功能，而 ES6 提供了几个新特性，显著提高了元编程能力。

从匿名函数的函数名推导，到提供了构造器调用方式这样的信息的元属性，你可以比过

去更深入地查看程序运行时的结构。通过公开符号可以覆盖原本特性，比如对象到原生类型的类型转换。代理可以拦截并自定义对象的各种底层操作，`Reflect` 提供了工具来模拟它们。

特性测试，甚至可以测试像尾递归优化这样微妙的语义特性，把元编程的焦点从你的程序转移到 JavaScript 引擎功能本身。通过更多地了解环境能力，你的程序可以在运行时调整自己达到最优效果。

应该使用元编程吗？我的建议是：首先应将重点放在了解这个语言的核心机制到底是如何工作的。而一旦你真正了解了 JavaScript 本身的运作机制，那么就是开始使用这些强大的元编程能力进一步应用这个语言的时候了。

第 8 章

ES6 之后

写作本部分的时候，ECMA 即将对 ES6（ECMAScript 2015）最终草案的批准进行正式投票。但尽管 ES6 还正在定案，TC39 委员会已经开始进行 ES7/2016 及后续特性的紧张工作了。

在第 1 章已经讨论过，我们可以预见 JavaScript 的发展节奏将要从每隔几年更新一次进化到每年一个正式版本更新（因此基于年度命名）。这将从根本上改变 JavaScript 开发者学习和追随这门语言发展进度的方式。

但更重要的是，实际上委员会将会以特性为单位工作。一旦某个特性标准完成，并且在几个浏览器通过实现测试了思路，这个特性就被认为足够稳定可以使用了。这强烈鼓励我们一旦某个特性可用就采用这个特性，而不是等待官方标准投票。如果你还没有开始学习 ES6，那么可就错过上船的时间了！

编写本部分时，可以在这里（https://github.com/tc39/ecma262#current-proposals）看到未来的提案及其状态。

在新特性还没有被需要支持的所有浏览器都实现的情况下，transpiler 和 polyfill 是我们迁移到新特性的桥梁。Babel、Traceur 和其他几个主要 transpiler 已经支持一些极可能确定的后 ES6 特性了。

认识到这一点，就会明白现在已经是开始了解这些特性的时候了。我们来学习吧！

> 这些特性还处于不同的开发阶段。虽然它们很可能会确定下来，并且将类似于本章所述，但不要把这一章的内容全盘接受。在未来的版本中，这一章内容会随着这些（以及其他！）特性的最终确定而进化。

8.1　异步函数

在 4.2 节中，我们提到了一个关于直接在语法上支持这个模式的提案：生成器向类似运行器的工具 `yield` 出 promise，这个运行器工具会在 promise 完成时恢复生成器。让我们来简单了解一下这个提案提出的特性 `async function`。

回忆一下第 4 章里这个生成器的例子：

```
run( function *main() {
    var ret = yield step1();

    try {
        ret = yield step2( ret );
    }
    catch (err) {
        ret = yield step2Failed( err );
    }

    ret = yield Promise.all([
        step3a( ret ),
        step3b( ret ),
        step3c( ret )
    ]);

    yield step4( ret );
} )
.then(
    function fulfilled(){
        // *main()成功完成
    },
    function rejected(reason){
        // 哎呀，出错了
    }
);
```

提案的 `async function` 语法不需要 `run(..)` 工具就可以表达同样的流控制逻辑，因为 JavaScript 将会自动了解如何寻找要等待和恢复的 promise。

考虑：

```
async function main() {
    var ret = await step1();

    try {
        ret = await step2( ret );
    }
    catch (err) {
        ret = await step2Failed( err );
    }

    ret = await Promise.all( [
```

```
        step3a( ret ),
        step3b( ret ),
        step3c( ret )
    ] );

    await step4( ret );
}

main()
.then(
    function fulfilled(){
        // main()成功完成
    },
    function rejected(reason){
        // 哎呀，出错了
    }
);
```

我们没有使用 `function *main() {..` 声明，而是使用了 `async function main() {..` 形式。而且，没有 `yield` 出一个 promise，而是 `await` 这个 promise。调用来运行函数 `main()` 实际上返回了一个可以直接观察的 promise。这和从 `run(main)` 调用返回的 promise 是等价的。

看到这种对称性了吗？ `async function` 本质上就是生成器 + promise + `run(..)` 模式的语法糖；它们底层的运作方式是一样的！

如果你是一个 C# 开发者，那么一定很熟悉这个 `async/await` 模式，因为这个特性就是直接来自于 C# 的对应特性。很高兴看到语言特性收敛。

Babel、Traceur 和其他 transpiler 都已经对当前状态的 `async function` 提供了早期支持，所以已经可以开始使用这个特性了。但在下一小节中，我们将会介绍为什么现在有点为时尚早。

> 还有一个对 `async function*` 的提案，可以称之为“异步生成器”。你可以在同一段代码中既 `yield` 又 `await`，甚至可以把这两个运算放在同一个语句：`x = await yield y`。这个“异步生成器”提案似乎更不稳定——具体说，它的返回值还没有完全确定。有些人认为返回值应该是一个 observable，有点类似于一个迭代器和一个 promise 的合并。目前我们不会深入探讨这个主题，但会对它保持关注。

警告

`async function` 有一个没有解决的问题，因为它只返回一个 promise，所以没有办法从外部**取消**一个正在运行的 `async function` 实例。如果这个异步操作的资源紧张，那么可能会引起问题，因为一旦你确认不需要结果就会想要释放资源。

举例来说：

```
async function request(url) {
    var resp = await (
        new Promise( function(resolve,reject){
            var xhr = new XMLHttpRequest();
            xhr.open( "GET", url );
            xhr.onreadystatechange = function(){
                if (xhr.readyState == 4) {
                    if (xhr.status == 200) {
                        resolve( xhr );
                    }
                    else {
                        reject( xhr.statusText );
                    }
                }
            };
            xhr.send();
        } )
    );

    return resp.responseText;
}

var pr = request( "http://some.url.1" );

pr.then(
    function fulfilled(responseText){
        // ajax成功
    },
    function rejected(reason){
        //哎呀，出错了
    }
);
```

我给出的这个 `request(..)` 有点像最近提出要集成到 Web 平台上的 `fetch(..)` 工具。那么问题来了，如果你想要用 `pr` 值以某种方法指示取消一个长时间运行的 Ajax 请求会怎样呢？

Promise 是不可取消的（至少在编写本部分的时候是如此）。和很多人一样，我的看法是它们永远不应该被取消（参见本系列《你不知道的 JavaScript（中卷）》第二部分）。而且即使它有一个 `cancel()` 方法，就一定意味着调用 `pr.cancel()` 应该把取消信号一路沿着 promise 链传播回到 `async function` 吗？

这个争论有以下几个可能的解决方案：

- `async function` 根本不能被取消（现状）；
- 可以在调用异步函数的时候传入一个“取消令牌”；
- 返回值变成一个新增的可取消 promise 类型；
- 返回值变成某种非 promise 的东西（比如，observable，或者支持 promise 和取消功能的控制令牌）。

编写本部分时，async function 返回普通 promise，所以返回值不太可能会彻底改变。但是判断最终如何发展还为时过早。我们对这个讨论保持关注吧。

8.2 Object.observe(..)

Web 前端开发的圣杯之一就是数据绑定——侦听数据对象的更新，同步这个数据的 DOM 表示。多数 JavaScript 框架都为这类操作提供了某种机制。

可能在后 ES6，我们将会看到通过工具 Object.observe(..) 直接添加到语言中的支持。本质上说，这个思路就是你可以建立一个侦听者（listener）来观察对象的改变，然后在每次变化发生时调用一个回调。例如，你可以据此更新 DOM。

你可以观察的改变有 6 种类型：

- add
- update
- delete
- reconfigure
- setPrototype
- preventExtensions

默认情况下，你可以得到所有这些类型的变化的通知，也可以进行过滤只侦听关注的类型。

考虑：

```
var obj = { a: 1, b: 2 };

Object.observe(
    obj,
    function(changes){
        for (var change of changes) {
            console.log( change );
        }
    },
    [ "add", "update", "delete" ]
);

obj.c = 3;
// { name: "c", object: obj, type: "add" }

obj.a = 42;
// { name: "a", object: obj, type: "update", oldValue: 1 }

delete obj.b;
// { name: "b", object: obj, type: "delete", oldValue: 2 }
```

除了主要的 "add"、"update" 和 "delete" 变化类型：

- 如果一个对象通过 Object.defineProperty(..) 重新配置这个对象的属性，比如修改它的 writable 属性，就会发出 "reconfigure" 改变事件。参见本系列《你不知道的 JavaScript（上卷）》第二部分可以获取更多信息；
- 如果一个对象通过 Object.preventExtensions(..) 变为不可扩展，就会发出 "prevent Extensions" 改变事件。

因为 Object.seal(..) 和 Object.freeze(..) 也都意味着 Object.preventExtensions(..)，所以它们也会发出相应的改变事件。另外，对象的每个属性都会发出 "reconfigure" 改变事件。如果一个对象的 [[Prototype]] 改变，或者通过 __proto__ setter 来设置，或者使用 Object.setPrototy peOf(..) 来设置，都会发出 "setPrototype" 改变事件。

注意，这些改变事件会在改变发生后立即发出。不要把这一点和代理混淆（参见第 7 章），代理是可以在动作发生之前拦截的。对象观察支持在变化（或一组变化）发生后响应。

8.2.1　自定义改变事件

除了前面 6 类内置改变事件，你也可以侦听和发出自定义改变事件。

考虑：

```
function observer(changes){
    for (var change of changes) {
        if (change.type == "recalc") {
            change.object.c =
                change.object.oldValue +
                change.object.a +
                change.object.b;
        }
    }
}

function changeObj(a,b) {
    var notifier = Object.getNotifier( obj );
    obj.a = a * 2;
    obj.b = b * 3;

    // 把改变事件排到一个集合中
    notifier.notify( {
        type: "recalc",
        name: "c",
        oldValue: obj.c
    } );
}

var obj = { a: 1, b: 2, c: 3 };

Object.observe(
    obj,
```

```
        observer,
        ["recalc"]
    );

    changeObj( 3, 11 );

    obj.a;        // 12
    obj.b;        // 30
    obj.c;        // 3
```

改变集合（"recalc" 自定义事件）已经排入队列准备发送给观测者，但是还没有发送，因此 obj.c 的值仍然是 3。

默认情况下，改变会在当前事件循环的最后发送（参见本系列《你不知道的 JavaScript（中卷）》第二部分）。如果你想要立即发送，可以使用 Object.deliverChangeRecords(observer)。一旦改变事件发送后，你就可以看到 obj.c 如预期地更新为：

```
obj.c;        // 42
```

在前面的例子中，我们用完成改变事件记录来调用 notifier.notify(..)。还有一种改变记录入队的方式是使用 performChange(..)，这会把指定事件类型从其余事件记录属性中分离出来（通过函数回调）。考虑：

```
notifier.performChange( "recalc", function(){
    return {
        name: "c",
        // this就是在观察之中的对象
        oldValue: this.c
    };
} );
```

某些情况下，这种关注分离可能会更干净地映射到你的使用模式。

8.2.2 结束观测

就像普通的事件侦听器一样，你可能希望停止观测一个对象的改变事件。为此，可以通过 Object.unobserve(..) 来实现。

举例来说：

```
var obj = { a: 1, b: 2 };

Object.observe( obj, function observer(changes) {
    for (var change of changes) {
        if (change.type == "setPrototype") {
            Object.unobserve(
                change.object, observer
            );
```

```
                break;
            }
        }
    } );
```

在这个小例子中，我们侦听改变事件，直到看到 "setPrototype" 事件发生，然后就停止观察任何新改变事件。

8.3 幂运算符

有提案提出为 JavaScript 新增一个运算符用于执行幂运算，就像 Math.pow(..) 一样。考虑：

```
var a = 2;

a ** 4;        // Math.pow( a, 4 ) == 16

a **= 3;       // a = Math.pow( a, 3 )
a;             // 8
```

** 实际上和 Python、Ruby、Perl 以及其他一些语言中的同名运算符一样。

8.4 对象属性与 ...

我们在 2.5 节已经看到，... 运算符展开和收集数组的用法很直观，那么对于对象呢？

本考虑在 ES6 中支持这个功能，但已经被推迟到了 ES6 之后（也就是"ES7"或者"ES2016"或者……）。下面是它在"ES6 之后"的时代中可能的工作方式：

```
var o1 = { a: 1, b: 2 },
    o2 = { c: 3 },
    o3 = { ...o1, ...o2, d: 4 };

console.log( o3.a, o3.b, o3.c, o3.d );
// 1 2 3 4
```

... 运算符可能也会用于把对象的解构属性收集到一个对象：

```
var o1 = { b: 2, c: 3, d: 4 };
var { b, ...o2 } = o1;

console.log( b, o2.c, o2.d );     // 2 3 4
```

在这里，...o2 把解构的 c 和 d 属性重新收集回到 o2 对象（o2 没有 o1 中的 a b 属性）。

再次声明，这只是 ES6 之后的在考虑之中的提案。但是如果实现的话会很酷。

8.5 Array#includes(..)

JavaScript 开发者需要执行的一个极其常见的任务就是在值数组中搜索一个值。一直以来实现这个任务的方法是：

```
var vals = [ "foo", "bar", 42, "baz" ];

if (vals.indexOf( 42 ) >= 0) {
    // 找到了！
}
```

使用 `>= 0` 检查的原因是，如果找到的话 `indexOf(..)` 返回一个 `0` 或者更大的数字值，如果没有找到就会返回 `-1`。换句话说，我们是在布尔值上下文中使用返回索引的函数。因为 `-1` 为真而不是假，所以需要更多的手动检查。

在本系列《你不知道的 JavaScript（中卷）》第一部分中，探讨了另外一种我更偏爱的模式：

```
var vals = [ "foo", "bar", 42, "baz" ];

if (~vals.indexOf( 42 )) {
    // 找到了！
}
```

这里的 ~ 运算符把 `indexOf(..)` 返回值规范为更适合强制转换为布尔型的值范围。也就是说，`-1` 产生 `0`(假)，所有其他值产生非 0 值 (真)，这正是判断是否找到这个值所需的。

我认为这是一个改进，然而其他人强烈反对。但是，没有人认为 `indexOf(..)` 的搜索逻辑是完美的。比如，它无法找到数组中 `NaN` 值。

于是出现了一个获得了大量支持的提案，提出增加一个真正返回布尔值的数组搜索方法，称为 `includes(..)`:

```
var vals = [ "foo", "bar", 42, "baz" ];

if (vals.includes( 42 )) {
    // 找到了！
}
```

> `Array#includes(..)` 使用的匹配逻辑能够找到 `NaN` 值，但是无法区分 `-0` 和 `0`（参见本系列《你不知道的 JavaScript（中卷）》第一部分）。如果你不关心程序中的 `-0` 值，那么这可能就是你所需要的。如果你确实在意这个 `-0` 值的话，那么你就需要实现自己的搜索逻辑，很可能是使用 `Object.is(..)` 工具（参见第 6 章）。

8.6 SIMD

我们在本系列《你不知道的 JavaScript（中卷）》第二部分中详细介绍了单指令多数据（Single Instruction, Multiple Data，SIMD），但是值得在这里简单说明一下，因为这很可能会是接下来出现在未来 JavaScript 中的特性之一。

SIMD API 暴露了可以同时对多个数字值运算的各种底层（CPU）指令。比如，你可以指定两个**向量**，其中分别有 4 个或 8 个数字，把它们的对应元素一次全部相乘（数据并行！）。

考虑：

```
var v1 = SIMD.float32x4( 3.14159, 21.0, 32.3, 55.55 );
var v2 = SIMD.float32x4( 2.1, 3.2, 4.3, 5.4 );

SIMD.float32x4.mul( v1, v2 );
// [ 6.597339, 67.2, 138.89, 299.97 ]
```

除了 `mul(..)`（相乘）之外，SIMD 还会包含其他几个运算，比如 `sub()`、`div()`、`abs()`、`neg()`、`sqrt()` 以及很多其他运算。

对于下一代高性能 JavaScript 应用来说，并行数学运算是很关键的。

8.7 WebAssembly (WASM)

在本部分即将完成的时候，Brendan Eich 对 WebAssembly（WASM）的最新声明，可能会对 JavaScript 的未来发展路线产生重大影响。这里我们无法详细介绍 WASM，因为在编写本部分的时候它还处于刚刚开始的阶段。但如果不简要介绍一下这一主题，本部分就是不完整的。

最近（以及不久的将来）JavaScript 语言设计修改上的最大压力之一就是需要成为更适合从其他语言（比如 C/C++、ClojureScript 等）变换 / 交叉编译的目标语言。显然，代码作为 JavaScript 运行的性能问题一直是一个主要关注点。

本系列《你不知道的 JavaScript（中卷）》第二部分中介绍过，几年前一组 Mozilla 开发者为 JavaScript 引入了一个新思路，称为 ASM.js。ASM.js 是合法 JavaScript 的一个子集，这个子集最严格地限制了那些使得 JavaScript 引擎难于优化的行为。结果就是兼容 ASM.js 的代码运行在支持 ASM 的引擎上时效率有巨大的提升，几乎与原生优化的等价 C 程序相当。很多人把 ASM.js 看作是高性能要求 JavaScript 应用使用 JavaScript 的最可能支柱。

换句话说，浏览器中运行代码的所有路径都通过 JavaScript。

直到 WASM 发布之前，是这样的。WASM 为其他语言在浏览器运行时环境中运行提供了一条新路径，不需要先通过 JavaScript。本质上说，如果 WASM 发布，JavaScript 引擎将

会获得执行二进制格式代码的新能力，这种格式某种程度上类似于字节码（bytecode，就像 JVM 上运行的那样）。

WASM 提出了一种代码的高度压缩 AST（语法树）二进制表示格式，然后可以直接向 JavaScript 引擎发出指令，而它的基础结构，不需要通过 JavaScript 解析，甚至不需要符合 JavaScript 的规则。像 C 或 C++ 这样的语言可以被直接编译为 WASM 格式而不是 ASM.js，这样通过跳过 JavaScript 解析会获得额外的速度优势。

WASM 的近期目标是与 ASM.js 和真正 JavaScript 相当。但最终的预期是，WASM 将会增加新功能，而这些新功能是超出 JavaScript 所能做的。比如像线程这样的激进功能给 JavaScript 带来了很大压力——这个改变将会给整个 JavaScript 生态系统带来巨大震撼——将很可能会成为一个 WASM 扩展，缓解 JavaScript 本身的修改压力。

实际上，这个新的发展图景为很多语言打开了新的道路，使其能够进入 Web 运行时。对于 Web 平台来说，这是一个令人激动的新特性。

对于 JavaScript 来说这意味着什么？ JavaScript 将会变得无关紧要或者“死去”吗？绝对不会！看起来在以后的几年里，ASM.js 不会有太大的发展了，但在 Web 平台中 JavaScript 的主体还是非常安全的。

WASM 的支持者认为，WASM 的成功将意味着 JavaScript 的设计可以免于被不现实的需求撕裂的压力。重点是，对于应用中的高性能部分 WASM 是更好的目标，可以用其他多种语言编写。

有趣的是，JavaScript 是未来不太可能转化为 WASM 的语言之一。未来的修改可能会刻划出 JavaScript 的一个适合于转化为 WASM 的子集，但是这条发展路径的优先级似乎并不高。

尽管 JavaScript 很可能不会转化为 WASM，但是 JavaScript 代码和 WASM 代码将能够最大程度地交互，就像现在的模块交互一样自然。你可以设想调用像 `foo()` 这样的 JavaScript 函数，而实际上调用的是一个同名的能够在你的其余 JavaScript 的限制之外良好运行的 WASM 函数。

当下用 JavaScript 编写的代码将可能继续用它编写，至少在可见的未来是这样。transpile 到 JavaScript 的东西将可能最终考虑使用 WASM 替代。对于那些性能要求极高，不能容忍多层抽象的功能，最有可能的选择是寻找合适的非 JavaScript 语言编写，然后以 WASM 为目标。

这个转变可能会比较缓慢，需要几年才能完成。WASM 进入所有主流浏览器平台可能至少也需要数年。同时，WASM 项目（https://github.com/WebAssembly）已经有一个早期的 polyfill 对其基本宗旨提供了概念证明。

但随着时间的发展，也随着 WASM 学到更多非 JavaScript 技巧，很可能当前一些 JavaScript 的东西会被重构为以 WASM 为目标的语言。举例来说，框架、游戏引擎以及其他常用工具中性能敏感的部分都可能从这样的转变中获益。在自己的 Web 应用中使用这些工具的开发者很可能不会注意到使用和集成过程中的差别，只会自动受益于性能和功能的提高。

可以确定的是，随着时间的发展 WASM 会越来越真实，对 Javascript 的发展方向和设计的影响也会越来越大。这可能是“ES6 之后”中最值得开发者关注的重要主题之一。

8.8 小结

如果说本质上本系列的其他几本书都是提出这个挑战：“你（可能）不（像你以为的那么）懂 JavaScript”，那么本部分就是在说：“你不再懂 JavaScript 了”。本部分覆盖了这个语言大量的 ES6 新主题，这是这个语言的令人激动的新特性和范式，将会永久地改进 JavaScript 程序。

但 ES6 并不是 JavaScript 的终结。还早得很呢！在“ES6 之后”这段时间已经出现了大量处于各种开发阶段的新特性。在这一章里，我们简单了解了那些在不久的将来很可能进入 JavaScript 的新特性。

`async function` 是建立在生成器 + promise 模式（参见第 4 章）之上的强大的语法糖。`Object.observe(..)` 为观察对象改变事件增加了直接的原生支持，这对于实现数据绑定很重要。幂运算符 `**`、针对对象属性的 `...` 以及 `Array#includes(..)` 都是对现有机制简单但有用的改进。最后，SIMD 把高性能 JavaScript 的革命带入一个新时代。

听起来像陈词滥调，但 JavaScript 的未来是光明的！这个系列，特别是本书的这一部分，已经把挑战放在了读者的面前。你还在等什么？是时候开始学习和探索了！

YOU DON'T KNOW JS

"当你努力理解自己所写的代码时，你不但创造了更好的工作成果，也提升了自己的能力。这些代码不再只是你的工作，而是成了你的手艺。这就是我喜爱本书的理由。"

——**Jenn Lukas**
前端顾问

"我认为没有比这更全面、更深入的ES6参考书了。"

——**Angus Croll**
Twitter工程师

你不知道的JavaScript 下卷

JavaScript语言有很多复杂的概念，但却用简单的方式体现出来（比如回调函数），因此，JavaScript开发者无需理解语言内部的原理，就能编写出功能全面的程序。然而，JavaScript的这些复杂精妙的概念才是语言的精髓，即使是经验丰富的JavaScript开发者，如果没有认真学习，也无法真正理解语言本身的特性。正是因为绝大多数人不求甚解，一遇到出乎意料的行为就认为是语言本身有缺陷，进而把相关的特性加入黑名单，久而久之就排除了这门语言的多样性，人为地使它变得不完整、不安全。

"你不知道的JavaScript"系列就是要让不求甚解的JavaScript开发者迎难而上，深入语言内部，弄清楚JavaScript每一个零部件的用途，轻松理解前端圈里出现的各种技术、框架和流行术语。本书介绍了该系统的两个主题："起步上路"以及"ES6及更新版本"。

作者介绍

Kyle Simpson，推崇开放的互联网，对JavaScript、HTML5、实时/端对端通信和Web性能有深入研究。他是技术书作家、技术培训师、讲师和开源社区的活跃成员。

封面设计：Karen Montgomery Randy Comer 张健

O'REILLY®
oreilly.com
YouDontKnowJS.com

图灵社区：iTuring.cn
热线：(010)51095186转600

分类建议 计算机／程序设计／JavaScript

人民邮电出版社网址：www.ptpress.com.cn

O'Reilly Media, Inc.授权人民邮电出版社出版

ISBN 978-7-115-47165-9

ISBN 978-7-115-47165-9
定价：79.00元

看完了

如果您对本书内容有疑问，可发邮件至 contact@turingbook.com，会有编辑或作译者协助答疑。也可访问图灵社区，参与本书讨论。

如果是有关电子书的建议或问题，请联系专用客服邮箱:
ebook@turingbook.com。

在这可以找到我们:

微博 @图灵教育：好书、活动每日播报
微博 @图灵社区：电子书和好文章的消息
微博 @图灵新知：图灵教育的科普小组
微信 图灵访谈：ituring_interview，讲述码农精彩人生
微信 图灵教育：turingbooks